

# 1. Introduction

*Like angels stopped upon the wing by sound  
Of harmony from Heaven's remotest spheres.*

– Wordsworth: *The Prelude*

## 1.1. The Two Cultures and the Great Theme

In *The Music of The Spheres*<sup>1</sup> Jamie James writes of music and science that “at the beginning of Western civilisation¼ the two were identified so profoundly that anyone who suggested that there was any essential difference between them would have been considered an ignoramus.” This is in stark contrast to today when anyone suggesting that they have anything in common “... runs the risk of being labelled a philistine by one group and a dilettante by the other and, most damning of all, a popularizer by both.”

The Great Theme beloved of the early philosopher scientists of a universe of perfect order in which everything has a purpose and a place, a universe whose very fabric sounded to continual heavenly music (which music obeyed the beautiful rules of the mathematics of Pythagoras and Plato) was discarded over the years of the Renaissance and into the Age of Reason. Though many present-day scientists have a great appreciation of the arts, those involved in the humanities tend to eschew the

---

<sup>1</sup> James, J., *The Music of the Spheres*, New York: Springer-Verlag, 1993.

cold empiricism of science. This is the age of C.P. Snow's *Two Cultures*<sup>2</sup>, which James describes as a "*psychotic bifurcation*". James elaborates:

"In the modern age it is a basic assumption that music appeals directly to the soul and bypasses the brain altogether, while science operates in just the reverse fashion, confining itself to the realm of pure ratiocination and having no contact at all with the soul. Another way of stating this duality is to marshal on the side of music Oscar Wilde's dictum that 'All art is quite useless,' while postulating that science is the apotheosis of earthly usefulness, having no connection with anything that is not tangibly of this world."

Despite centuries of divergence, there are indications that some, at least, are starting to build bridges between the cultures again. In Douglas Adams' comedy novel *Dirk Gently's Holistic Detective Agency*<sup>3</sup> the character MacDuff attempts to produce music from mathematical representations of the dynamics of swallows in flight. In a marvellous reiteration of the Great Theme, Adams writes of MacDuff's belief that "*if ¼ the rhythms and harmonies of music which he found most satisfying could be found in, or at least derived from, the rhythms and harmonies of naturally occurring phenomena, then satisfying forms of modality and intonation should emerge naturally as well.*"

Although a single work of fiction is not sufficient evidence of a swing away from The Two Cultures, Adams' thinking is indicative of a growing trend in the computer science research community. Adams, best known for his popular book, radio, and television series, *The Hitchhiker's Guide to the Galaxy*, is a commentator on the development of computer technology and its place in society. In his banquet speech to the Brown/MIT Vannevar Bush Symposium (12-13 October, 1995) [144] he alluded to The Two Cultures with the analogy of the two Amazonian tributaries, one of white water and the other of black, that join but do not merge. The streams travel in parallel for many miles before the waters finally blend.

It is questionable whether MacDuff's beliefs are an intentional move by Adams towards re-establishing the philosophy of the early scientists among modern re-

---

<sup>2</sup> *The Two Cultures* refers to the existence of two separate cultures with little contact between them; one culture is based on the humanities and the other on the sciences. The phrase gained popularity after C.P. (Lord) Snow's Rede Lecture, later published as *The Two Cultures and the Scientific Revolution* (1959). c.f. Matthew Arnold's *Culture and Anarchy* (1869) and his Rede Lecture *Literature and Science* (1882) [27].

<sup>3</sup> Adams, D., *Dirk Gently's Holistic Detective Agency*, Pan, 1988.

searchers; however, the fact remains that computer scientists (unwittingly or not) are making increasing use of artistic forms (be they aural or visual) in their work.

## 1.2. Human-computer interaction

Perhaps the most popular area for employing such 'new' ideas is the field of human-computer interaction (HCI). Since the introduction of the visual display unit (VDU) much research effort has gone into finding new and better ways to maximise the use of the video channel. Developers have been quick to maximise the use of graphical display capabilities from the use of menus on character-based displays to the visually impressive graphical user interfaces (GUI) of today.

Alongside the development of visual presentation, psychologists spent much time analysing and studying the effects on computer users of different methods of information display. This has led to a well-established body of research into the exploitation of the visual medium as a means of interfacing the increasingly powerful and sophisticated computer technology with a more discerning and expectant user community.

The research community has been slow to recognise sound as a useful carrier of information in the world of software development. This can be partially attributed to the relatively late arrival of widely available and cheap sound generating devices for computers. While graphical capability has been available for many years, cheap audio facilities are much more recent additions to the personal computer. By contrast, early computer users often used sound. The most oft-repeated anecdote is that of the early programmer who tuned an AM radio to pick up the radio interference put out by the computer. By listening to the patterns of sounds in the interference they learnt to monitor CPU behaviour and even to identify errant program behaviour.

The lack of standards for sound equipment also acts as a deterrent. The Musical Instrument Digital Interface (MIDI) specification [119] at least goes some way to providing a common language, although it is oriented towards the communication of musical data and not that of sound generally.

By the time affordable sound generating equipment became available to the average computer user the study of the visual medium was well advanced. For largely technological reasons the human-computer interface has, from the start, been al-

most entirely visual in its construction. This may have helped to foster the belief that the computer user is more naturally a visile [44] or one whose mental imagery takes a visual form. Such people will tend to say that they ‘see what you mean’. With the advances in display technology came an inertia that led to an increasing bias towards visual interfaces. This is reflected in the natural language of those cultures that rely on the written word for communication (particularly English), which by using words like ‘imagery’ to describe mental processes shows an inclination towards visual metaphors for the explanation of ideas. The very act of thinking is defined using the visually oriented word ‘imagine’. English offers very few tools to describe the mental processes of audiles who tend to say that they ‘hear what you mean’. Contrast this with cultures that have an oral tradition– they are much more multi-sensory in their communication. Somers [147] comments:

Speakers in non-literate cultures, including children in all cultures who have not yet learned to read, tend to use many inflections and gestures. But as people become educated in literate cultures they are often taught to “modulate” their vocal inflections, stand still as they talk, and not use gestures. Thus speech becomes reduced to the single element which can be coded by writing or printing: the meaning of the words themselves.

Given our tendency to represent ideas as images in our minds and the historical development of visual display hardware, it seems reasonable that the emphasis in software development and human computer interface design has focused predominantly on the ocular.

### 1.3. The programming problem

The sophistication of personal-computer software has increased enormously over the last ten to fifteen years. Despite the abundance of tools designed to make the generation of working applications easier and easier, such as Visual Basic, fourth generation languages, and the macro-language add-ons to integrated office packages, programming remains a core skill in most computing curricula. Whatever the sophistication of the development environments, for the foreseeable future at least, programming skills will continue to be in demand.

One of the largest problems novice programmers face is understanding the code they have written. Debugging does not work well visually. Program events are in the time domain whereas visual mappings give predominantly spatial representations.

Visual techniques give us good descriptions of spatial relations and structural details (just like Fourier analysis does for sound waves), but do not naturally represent temporal details. Using sound might present us with a complementary modality which will increase the diagnostic tools available by giving a temporal view of software (as the wave-form plot does for a sound wave). The debugging task is made harder because the novice does not yet possess the repertoire of skills needed to diagnose, or even spot, the symptoms of an incorrectly functioning program. Too often, the novice is more concerned with simply writing source code that will compile without syntax errors than with ensuring that the code is a correct realisation of the specification. The problem is exacerbated when the novice is faced with maintaining, or even debugging, a program written by a third party. Techniques that a skilled programmer uses to understand and debug a program are not known or employed by the novice [71].

For example, Nanja and Cook [123] showed that while experts try to understand a program before looking for bugs, novices tend to look immediately for candidate bug locations by searching the output for clues, recalling similar bugs and testing program states. Further, experts tend to read a program in terms of its execution order whereas novices will read the program text like a book from start to finish. Thus experts try to gain a high-level understanding first whilst novices use a bottom up approach to understanding the program. Ramalingam and Wiedenbeck [133] and Wiedenbeck et al. [160] found in studies of novices that when working with imperative languages (such as Pascal) subjects could demonstrate a lot of program-level knowledge but little domain-level knowledge. Conversely, they also found that when working with object-oriented languages (such as C++) novices were much more able to gain a high-level domain understanding. Vessey [151] found that novices failed to develop mental models of program structure and were thus unable to infer causal behaviour.

In a comparative study of novice and expert programmers, Allwood and Björhag [2] observed that novices tend to make many more false identifications of bugs than experts. They asserted that for novices debugging has the character of a problem-solving process. Novices, they said, possess less domain-relevant knowledge and less of it automatised. For example, novices tend to lack specific associations between specific symptoms such as error messages [2].

Ormerod [126] identified two important differences between expert and novice programmers. First, novices tend to view problems as belonging to a specific domain (e.g. payroll, banking, or science) whereas experts sort problems according to algorithmic type. Secondly, novices tend to work forwards from a problem, line-by-line until they arrive at a solution. Experts, on the other hand, work backwards, decomposing the problem into manageable chunks.

Personal experience in teaching programming at the introductory level shows that while students may understand that a program does not work, and may even know which general part of the program is at fault, they still have insufficient understanding of how the program executes and thus cannot see why the bug is manifesting itself. Our experience also shows that at the early stages, novices can spend a long time trying to debug a program they have written, even when that program is actually very short (fewer than twenty lines of code). Much of this may be due to the tendency of novices to use their natural language understanding of a program when solving a problem [25]; they have not yet learned the difference between natural language and the more precise logic of programming languages. This is especially so when trying to write code involving compound Boolean expressions. The way that words like 'and' and 'or' are used in natural language, to the novice, can make their precise use in languages such as Pascal appear counter-intuitive. A common novice error of this type is to write `WHILE (name <> 'ZZZZZ') AND (mark <> 0) DO` instead of `WHILE (name <> 'ZZZZZ') OR (mark <> 0) DO` for a loop that is to terminate only when both variables (*name* and *mark*) contain their terminal values [138].

The problems associated with debugging programs have long been recognised and many tools to assist in the process have been developed. Common features of debuggers include:

- Step and trace facilities to allow execution of a program line by line.
- Break points that enable programs to be run up to a user-defined point at which time execution halts and program variables can be inspected.
- Monitoring facilities that let the programmer examine the values of variables, expressions and data structures during the course of program execution.
- Browsers that show what procedures call other procedures and, in some cases, what procedure *called* a particular subprogram.

At their simplest, debuggers use basic textual output, possibly in a windowing environment and often within the interface of the compiler, e.g. Borland's Turbo Pascal. At the other end of the scale come the tools that employ sophisticated graphical displays and visualisation techniques in an attempt to assist the programmer in understanding the execution of his program. Parallel programming environments in particular have benefited from this approach as the debugging problem is made harder by the concurrent execution of several programs or subprograms.

One danger faced by novices is that they will be put off programming by early failures and an inability to understand relatively straightforward programs. It is desirable to make the transition from absolute beginner to competent learner as painless and as confidence-enhancing as possible. Any tool that assists novices in understanding their own work and that of others and thus in writing better programs and finding bugs more easily is of benefit.

In his Ph.D. thesis, Moan [120] asserts that the use of colours and graphics to display conceptual models of programming features will improve students' understanding of programming languages. The use of visual display techniques has been well developed and some very sophisticated visualisation systems now exist. Whilst much effort has been, and continues to be, put into the development of visual techniques for enhancing program understanding, there is a potential assistive technology that has been largely ignored. The audio channel remains relatively little-researched for its potential for assisting in the programming and debugging process. It is only recently that this medium is beginning to be exploited by researchers and practitioners in HCI applications.

The reliance on visual-oriented interfaces produces its own stresses. It is very easy for screens to become cluttered with multiple information windows. Information in a particular window may become hidden by subsequent windows. A user cannot attend to the entire screen, especially in the case of multiple-windowed graphical user interfaces. Good design approaches can ameliorate some of these problems, but the use of visual approaches alone still creates problems for the blind and visually-impaired community. These users, reliant on screen readers (and in some cases, braille) cannot use graphical programming and debugging tools. The use

of screen reader technologies is made more difficult by the multi-window implementations of modern program development environments.

In *Dirk Gently's Holistic Detective Agency* MacDuff made himself wealthy by devising a spreadsheet program that allowed company accounts to be represented musically. MacDuff states sarcastically that the “*yearly accounts of most British companies emerged sounding like the Dead March from Saul.*” Science fiction often precedes science fact. Much of what writers like Arthur C. Clarke described in their works of fiction in the 1940s and 1950s their audience saw realised in the 1960s, '70s and '80s. Putting a man on the moon may have seemed a flight of fancy forty years ago; today it is an historical fact three decades old. Although Adams' idea of the musical spreadsheet may have seemed absurd in 1987, researchers like Kramer [94] have since reported the successful use of auditory mappings of stock market data to identify market trends.

There is much about the audio channel's ability to communicate useful information to the computer user to be explored. An attempt to show whether, using a technique known as program auralisation<sup>4</sup> [91], musical feedback is a useful mechanism in helping novice programmers debug their code forms the subject matter of this thesis.

## 1.4. Overview of thesis

This thesis investigates the use of music as a communication medium in the task of computer programming and debugging. It describes previous work carried out in the field of *program auralisation* and presents an argument for using musical frameworks and grammars. It then goes on to describe a system for auralising the Pascal code of novice programmers. Experiments to determine whether the system proved helpful to the novice in debugging his programs are then described and the results discussed. The thesis comprises 7 chapters, the first being this introduction.

Chapter 2 surveys the use of sound in various human-computer interaction applications such as sonically-enhanced interfaces, scientific visualisations and inspection of program state and behaviour. A taxonomy of sound-related applications in



the human-computer interface is described. In the light of this, the role of sound in the debugging process is discussed.

Chapter 3 argues for the use of music as an effective communication medium and discusses why it can be superior to other types of audio. Chapter 4 describes the CAITLIN system that was used as the main tool for obtaining empirical results to support the thesis. Heuristic evaluation was carried out and is described.

Chapter 5 discusses various music-theoretic, cognitive and empirically observed aspects of music and its wider use in the interface. Using this knowledge some organising principles for musical auralisations are proposed. A complete set of auralisation motifs is described, some of which make use of the organising principles and some of which do not. A recognition study involving advanced novice programmers is then discussed. Chapter 6 examines how novice programmers reacted to using the redesigned CAITLIN motifs in various bug location tasks.

Chapter 7 offers conclusions drawn from the evaluation of CAITLIN and discusses some of the possible implications of using musical auralisation techniques in education and in the workplace. Finally, recommendations for further work and future research are given.

---

<sup>4</sup> *Auralisation* is a recently coined word. There are several other such terms and phrases that relate to the use of sound in the interface. A glossary of such terms is provided appendix A to assist the reader.