Chapter 1

Introduction

1.1 The missing, dynamic half of the Internet story

1.1.1 A universal tool for sharing control?

The Internet is a vastly under-utilised resource. It is only used for half of the IT (Information Technology) story.

IT is defined as the use of technology to manage and process information. But information for what? The destination of this information is either man or machine—information that gives us knowledge, or information that controls how things work. We can call these two sides of IT knowledge-IT and control-IT. We experience them both every day. The Web is a prime example of knowledge-IT. How our mobile phone is programmed to work is an example of control-IT.

Ask anyone to describe the Internet as best they can in two words and they might say ‘sharing knowledge’. But it would be more accurate to say ‘sharing information’, since the Internet is, of course, just the global cabling used for IT. It’s just that all the principle ways we use the Internet—the Web, email and media streaming—happen to be examples of knowledge-IT.

Knowledge-IT is essentially a static medium, the role of IT being simply to present us with information in suitable visual or audible form. But control-IT is characterised by being dynamic, since information is used here for driving machines, devices and systems. This is the realm of software programming.

The Internet is the driving force in the IT industry, so why isn’t it known as a universal tool for sharing control in the way it is for knowledge? True, there are examples of specialised, one-off software applications interfacing with each other via the Internet, but there has yet to be any systematic and universal attempt to exploit the potential of the Internet for control-IT in the way we have seen it for knowledge-IT.

Arguably, control-IT has greater significance in our lives than knowledge-IT. This means that the Internet, for all the impact it has had, is yet to be used for even half of its potential.

1.1.2 Lessons of the Web

The success of the Web lies in the simplicity of its design. It has become so powerful and ubiquitous because it is built around the well-known principle that less is more.

The Web works so well because its design is highly standardised. The universal language of the Web is HTML, a simple, markup language\(^1\). Everyone using the Web runs the same application.

\(^1\)A markup language combines text and extra information about the text expressed in markup—special coded tags which are embedded in the text. HTML is the best known example.
program, the Web browser, for displaying the contents of Web pages written in HTML\textsuperscript{2}.

The Web’s emphasis on standardisation and simplicity encourages universal participation, not just in the use of Web services but in their creation too. A recent revolution in the direction of the Web is Web 2.0\textsuperscript{3} that encourages online collaboration and sharing among users through wikis.\textsuperscript{4} A typical example is the Wikipedia online encyclopedia [51], where people with specialised knowledge about a topic are encouraged to add to the encyclopedia’s database.

Why go to all the trouble of creating something yourself when you can get the whole world to help you?

1.1.3 Why the Internet revolution in control-IT is yet to happen

Consider how different this all is from how we approach control-IT. Unlike the Web, information sharing—and thus the freedom to change how things work—is not encouraged. Buy a mobile phone and that is how it will always work. Those responsible for creating its operating software are restricted to the very few who have the necessary, specialised programming skills and intimate knowledge of how the phone has been manufactured.

None of the principles that make the Web so successful—design simplicity, same tools for all, information sharing—are followed. It is almost as if a diametrically opposed design philosophy has been chosen.

\textsuperscript{2}The occasional need for embedding code written in another language into a Web page, such as a Java applet, shows that in practice HTML hasn’t been designed to be powerful enough to lay true claim to being the single, universal language of the Web.

\textsuperscript{3}Web 2.0, first coined in 2004, is a loose term used to describe a second generation of Web services that focus on user-generated content.

\textsuperscript{4}A wiki is a website that supports collaborative Web page authoring by allowing its users to add, remove, and modify its contents.
1.2 The Mesh

1.2.1 A parallel, dynamic world to the Web

The Web has shown us the benefits to be had by replacing top-heavy, traditional engineering solutions in knowledge-IT with simpler, universal and more flexible alternatives. This thesis discusses how a similar approach might be possible in control-IT.

Taking the Web model as a starting point, it proposes a new design philosophy that changes the way we traditionally think about software design and the art of making things work. It builds on this design philosophy to create a working concept demonstrator that demonstrates exactly how the system would work and be implemented using the Internet.

It introduces the concept of the The Mesh. In the same way that the Web seamlessly connects databases of the world to provide a global font of knowledge, the Mesh would connect software of the world to provide a global means of control. The word ‘mesh’ is apt, since it conveys the correct meaning, both in the static sense of a net and in the dynamic sense of the interplay between running machinery connected by meshing gear wheels.

![The Mesh](image)

Figure 1.1: The Mesh

In the same way that we use the Web to ‘find out about anything in the world’, we would use the Mesh to ‘control anything in the world’. The Mesh would embody all the successful, empowering features of the Web. Everyone would have a say in how things work, mirroring Web 2.0’s user-generated content but for software instead of media.

But many features of the Mesh would be unique. They would change the way we go about software design, leading to new opportunities for users, programmers and manufacturers alike. More than just a new technology, we might expect the Mesh to become a force for social change in much the same way that the Web has, with new models for cultural and commercial interaction involving the theme of control rather than knowledge.

As well as providing a universal tool for describing how a device works (from here on, the term ‘device’ is used collectively to include all programmable machines, electronic devices and systems), it would be possible to link all these devices in realtime over the Internet to create a world of interconnecting and interoperating devices. It would even be theoretically possible to network all
devices in the world so that they behaved as a single, globally distributed device.

1.2.2 Some practical examples

The myriad uses of the Web that have been developed since its inception are well known. Here are a few examples of the types of problem the Mesh might be able to address in the control of everyday consumer products and appliances:

“I hate resetting the clocks in appliances after a power cut. I never remember how to, since each procedure is different. Why can’t this tedious job be automated somehow? Think of the wasted time and effort this would save multiplied across the city!”

“I want all the fans in my house to work as one, whichever one I choose to operate. And I want them to switch on automatically on hot days. So what if they’ve been cheaply manufactured with no means of detecting the temperature!?”

“Why can’t I download software for my Nokia mobile phone that makes use of all its in-built hardware and changes it into a new type of device altogether? Maybe an emergency pager where even the incoming call procedures and keypad work differently.”

“All my electronic devices are so confusing to use, cluttered up with features I don’t need. Why can’t I choose the features I want like toppings for a pizza? Or maybe get someone, or something, to choose for me?”

“All this talk has given me a great idea for an e-business buying and selling DIY software. I’ll call it ‘ui4u’—user interfaces for you. Customers will get exactly what they want and I’ll make money on each user interface ‘topping’ that I trade.”

1.2.3 The Long Tail and how our consumer culture is changing

These examples show how an important use of the Mesh would be to customise how devices work to suit individual requirements on a level that has never been seen before, exactly in line with the way our economy and culture is changing.

The phrase The Long Tail describes this recent social phenomenon. First coined by Chris Anderson and discussed in his recent book The Long Tail: Why the Future of Business is Selling Less of More [3], it describes the shift from mass markets to niches. As the costs of production and distribution fall, especially online, there is less need to lump products and consumers into a one-size-fits-all mentality. People gravitate towards niches because they satisfy individual needs better.

Realisation of this social shift is fueling new ways of commercial interaction because:

- greater choice leads to greater demand
- people are often willing to pay a premium for goods and services that suit them better
- when summed, these niche markets often exceed the market share of the mainstream market.

In recognition of how this trend has captured popular imagination, Anderson was named as one of Time magazine’s top 100 influential people in the world for 2007.

Many examples of the Long Tail can be found in knowledge-IT, but the Mesh would take the Long Tail into the undiscovered territory of control-IT.
1.2.4 Current research areas working at the big picture level

The Mesh would encompass a number of established research areas working on the issue of control at this big picture level, including:

- universal usability
- ubiquitous computing
- interface agents
- grid computing.

Universal usability is about the way we, as users, control the devices in our lives. It is closely based on the universal design concept, defined by the Center for Universal Design [41] in North Carolina, USA as:

*The design of products and environments to be usable by all people, to the greatest extent possible, without the need for adaptation or specialized design. The intent of universal design is to simplify life for everyone by making products, communications, and the built environment more usable by as many people as possible at little or no extra cost. Universal design benefits people of all ages and abilities.*

![The universal design concept](image)

Figure 1.2: The universal design concept

Ubiquitous computing [44] seeks to embed small, networked computers into everyday objects to provide device interoperation and new means of control. For example, biometric sensors capable of automatically regulating the ambient environment might be woven into our clothing.

Interface agents [24] are autonomous software tools designed to take control of devices and make decisions on our behalf. They observe how we interact with a user interface and provide us assistance by direct manipulation of the interface [30]. This is the field of research that includes adaptive systems and intelligent user interfaces, where automated control is proactive as well as reactive.

Grid computing [17] addresses the issue of how best to manage and control software execution across a network in order to improve executional efficiency and make best use of available resources.
CHAPTER 1. INTRODUCTION

It is the idea of software concurrency on a large scale. Program threads run simultaneously on a number of disparate, geographically distributed devices communicating with each other over a network.

1.2.5 Taking a broader view

Though these research areas overlap to some extent, they all tend to focus on a different aspect of the control issue at the big picture level. For example, universal usability is primarily concerned with the needs of users. In ubiquitous computing, however, the emphasis is on hardware and how to make devices communicate with one another over a network.

Table 1.1 lists these research areas, their primary focus of interest and the main design challenge that each one faces.

<table>
<thead>
<tr>
<th>Research area</th>
<th>Focus</th>
<th>Main design challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal usability</td>
<td>User</td>
<td>How to make user interfaces uniquely customisable to suit individual needs and run on any platform?</td>
</tr>
<tr>
<td>Ubiquitous computing</td>
<td>Hardware</td>
<td>How to make devices of all description easy to interoperate over a network?</td>
</tr>
<tr>
<td>Interface agents</td>
<td>User</td>
<td>How to simplify the task of designing and building agents for many different types of device, and minimise the scrutability problem?</td>
</tr>
<tr>
<td>Grid computing</td>
<td>Software</td>
<td>How to break down a program into threads that are able to exchange information at any time while running concurrently on a network of processors of any device type?</td>
</tr>
</tbody>
</table>

Table 1.1: Current research areas working on the control issue at the big picture level

The Mesh takes a broader view of the control problem than any of these research areas: a vision in which the interactions between users, hardware and software would be globally networked to provide universal solutions in control. In doing this, the Mesh would seek to address many of the individual design challenges of these research areas.

Rather than seeing the problem to be solved as essentially one of user customisation, device interoperability or some other limited aim, the philosophy of the Mesh would be to provide a universal set of working methods for solving the problem at hand with whatever resources are available at the time. The Mesh would not be a networked solution per se. For example, the flexible nature of the Mesh would make it easy to customise the user interface for a stand-alone device (the goal of universal usability). But should that device gain an Internet connection, then even greater customisation would be attainable, since it would now be possible to link how the device works to the operation of other devices (the goal of ubiquitous computing).
1.3 Motivation and background

1.3.1 An ever-widening focus

The reasons for undertaking this PhD originate from early work in mobile phone design and the belief that there were ways of making mobile phones more adaptable in meeting the diverse needs of their users. While working on this problem, it became apparent that the solutions being developed would equally well apply to the control of any type of electronic device.

The opportunity arose of developing these ideas further in the supportive and intellectually rigorous environment of a PhD research project in universal usability at the University of Adelaide. It was only during this research that, once again, it was found that the solutions under development had wider application than originally intended.

1.3.2 An unorthodox approach

The reader might be forgiven for wondering whether this thesis is about user interface design, the development of a simple, general-purpose computer language, dynamically extensible software or a critique on current thinking in engineering design.

An engineering PhD usually focuses on a known problem and looks for either a new or better way to solve it. The vision is necessarily task-oriented and narrow. While this approach has been followed in the big picture sense of seeking a better way of doing things, the route taken has been unorthodox. Instead of the usual preoccupation with the nature of the problem to be solved, there has been a conscious attempt to step back and take a broader view of the problem solving process before jumping in to consider the specific, engineering design challenges.

For the author, this meant trying not to think like an engineer, at least initially—something that wasn’t hard given his lack of a formal engineering background! But it was realised that any insights would be of little value unless they could produce tangible results at the engineering coalface. Here was the particular value of undertaking the PhD in electrical and electronic engineering, and not in some other discipline less predisposed to critical review of the engineering realities.

1.3.3 Validation in the form of a concept demonstrator

The aim was to develop ideas to the point that they were clearly shown to be workable, practical and potentially useful. The means of doing this would be reasoned argument, backed up as far as possible by working examples in a concept demonstrator.

With so many ideas quickly being generated, it became impossible to work through each one fully, covering all the implied angles and potential uses. There was insufficient time to do this—and anyway, how could the take-up on a new idea and where it might eventually lead be reliably predicted? But it was felt that this would not be critical to the success of the work, and couldn’t reasonably be expected within the confines of a lone PhD research project. It would be sufficient to introduce each idea, demonstrate that its supporting mechanisms work and give at least some examples of its usefulness.

1.3.4 Answers looking for problems—not problems looking for answers

By avoiding having a specific goal in mind and concentrating on developing tools based on a different philosophy about design, the traditional approach for a PhD was turned on its head. The focus became answers looking for problems, not problems looking for answers. This made the
journey all the more interesting and potentially rewarding, since the final destination was never limited by predetermined expectations from the start.

It was realised that such an approach could not be expected to deliver a neatly packaged set of working methods, all of equal relevance and merit. But encouragement was taken from their overall cohesiveness in addressing a broad range of known problems, while at the same time pointing the way to new and interesting areas of application.
1.4 Aim of the thesis

1.4.1 Conceptual and technical validation of the Mesh

The aim of this thesis is to validate the concept of the Mesh, not just at the software engineering level but also to the extent that the Mesh would be of practical benefit in leading to useful applications capable of addressing real world problems.

The author felt that an emphasis on just one of these areas as the expense of the other would fail to do justice to the work undertaken, since the danger would then be for the reader either to be left with an understanding of the nuts and bolts of the Mesh but with no insight as to why the Mesh might be useful, or conversely, with a broad appreciation of the potential of the Mesh but in the vacuum of wondering whether it could actually be made to work when hard engineering realities were faced.
CHAPTER 1. INTRODUCTION

1.5 Literature review

1.5.1 Cutting across existing research boundaries

The nature of this research is very broad in application, touching on and contributing to many areas of research across the social/technological spectrum.

The main thrust of this research relates to the application of cutting edge and emerging technologies in large-scale control such as universal usability, ubiquitous computing, grid computing and adaptive systems. At the software engineering level, the research relates to a number of software design aspects that include embedded device control, concurrency management, aspect-oriented programming and dynamically extensible software.

The broad nature of this research is unusual in an engineering PhD, especially one with such a strong focus on software design. Accordingly, this literature review should be read as an introduction to the many and varied areas of interest that this research addresses. More detailed discussion of the literature is given in subsequent chapters when and where it applies to the particular aspect of the Mesh under discussion.

1.5.2 Using the power of abstraction

It is recognised that the only effective tool we have for combating complexity is the power of abstraction [20]. In being the outcome of a new philosophy about design (see section 1.8.4 on page 18), the Mesh owes much to the inspiration derived from observing how fields unrelated to engineering have successfully applied the abstraction principle to meet specific design challenges.

This research took particular note of the DNA model for encoding design, demonstrating the practical benefits to be had from biomimicry in which designs found in nature are studied and imitated to solve human problems [6].

1.5.3 Social and commercial aspects and implications

Mention was made in section 1.2.3 on page 4 of how the Mesh closely ties in with the way that our consumer culture is changing, as represented by the phenomenon of the Long Tail [3]. It has been an overriding aim of this research to extend the Long Tail into the, as yet, undiscovered territory of software design and the control of the devices and systems in our world.

In creating a control highway—the dynamic equivalent of the Internet’s current use as an information highway—the Mesh would emulate and build on many of the social and commercial strategies for information management that are currently driving Web development, evident in such Web sites as Google [15], Wikipedia [51], eBay [10] and YouTube [54]. An example of this is discussed in section 3.21.8 on page 118 where eBay’s peer review system for ensuring trader reliability is proposed as a useful model for ensuring the reliability of shareware in the Mesh.

The wide ranging nature of the capabilities of the Mesh and the many practical applications that it could be used for raise many social, commercial and financial issues needing to be explored further. Unfortunately, there has been insufficient time to do this, given the need to fully address the technological issues first.

1.5.4 User interface design

This research work has a strong focus on the body of research represented by the collective title of universal usability (see section 3.3.2 on page 41). Universal usability is, itself, closely aligned...
to the well-established, more general concept of universal design [41] that seeks to make products and environments more usable for all.

The premise of universal usability is to improve how the many devices and systems work in our lives to the benefit of all users, particular those often marginalised in our society by poor user interface design such as the disabled and elderly. Progress has been slow, in part because the work involves a disparate mix of disciplines including hardware design, computer science and psychology. But also, universal usability has had to face new design challenges posed by the shift to smaller platforms seen in mobile and hand-held devices like mobile phones where additional considerations need to be addressed such as limited space for physical controls and information display.

While fundamental usability principles are well understood, there are many complicating factors in user interface design. For example, even individual needs may not remain constant, changing according to situation or over time as we age [29]. Psychological factors can play a strong role too. Research has shown that if we like the appearance of a system we may be far more tolerant of any of its actual shortcomings [33].

It’s known that a one-size-fits-all approach in user interface design can never be the answer, since everyone has different needs [7] [49]. What is required are user interfaces that can more easily be customised and personalised to suit any user situation [1] [16] [35]. New methodologies are needed to reduce system complexity while expanding functionality [23] [38]. Flexibility is the key [37], but this goal is hard to achieve once a user interface is coded as a conventional, monolithic program. A common theme is to provide greater levels of abstraction in the design process, thereby allowing designers to focus on the relevant logical aspects while avoiding low-level detail [13].

Various attempts have been made to create a ‘universal user interface’ through the application of special mechanisms. Examples include providing alternative, standardised means of system operation through the provision of duplicated physical controls [27], special development tools designed to help the interface designer to create customised user interfaces [2], and the support for multi-level user interfaces targeted at different categories of users such as novices and experts [39]. Others have tried to address the problem by creating functional modules that can be assembled different ways to build up a final, working user interface [5] [18]. All these different design approaches have merit, but none so far have proved to be of universal application.

Flexibility in user interface design is conventionally seen as the process of increasing user options by packing more user procedures into the user interface. The problems that this can cause are well documented [4] [31] [32] [50]. The result is that the design process is reduced to the unenviable job of finding the best trade-off between conflicting needs for simplicity and flexibility [33].

One approach for achieving user customisation and personalisation is to use interface agents. Interface agents are autonomous software tools used in adaptive systems that make decisions on behalf of the user [24]. These tools monitor how the user interacts with a user interface and provide assistance by direct manipulation of the interface [30]. While there are many advantages in automating the customisation process, there is much ongoing debate on the advisability of using interface agents because they tend to violate good usability principles such as predictability and scrutability.

Section 3.13.4 on page 90 discusses how the need to write device-specific algorithms for modifying how devices work in adaptive systems could be completely replaced in the Mesh by the simpler option of a universal search engine for locating ready-made software solutions.

1.5.5 The rise of XML-based languages in user interface design

A recent trend has been the switch to the use of XML-based languages for creating adaptive user interface design systems capable of handling cross-platform applications. The usual approach taken
CHAPTER 1. INTRODUCTION

is to split a user interface into an XML-coded top half operated by the user, and a conventionally
coded bottom half running the device—the top half customizing device operation for a particular
user situation, modality or platform [7] [36].

Examples of these User Interface Description Languages (UIDLs) include: UIML (User Interface
Markup Language) [48], UsiXML (USer Interface eXtensible Markup Language) [47], XForms
(XML Format Web Forms) [53], XIML (EXtensible Interface Markup Language) [11] and XUL
eXtensible User Interface Language) [12].

There is considerable debate on which UIDL is the best. Trewin [43] believes that a UIDL should
be: applicable to any physical device or service, applicable to any context of use, personalisable,
flexible, extensible and simple.

For all their many advantages, UIDLs suffer from a number of fundamental limitations. They
do not attempt to address the whole universal usability problem, since they still heavily rely on the
use of conventional programming languages. They are also by nature strictly declarative, though
some would argue that there is no hard and fast distinction to be made between declarative and
imperative programming [26].

UIDLs only attempt to address the specific problem of user interface design. They are not able
to offer the programming capabilities of conventional, abstract languages such as Java or C++. In
being built on a priori information about how a user interface should be designed to work, each
UIDL has a particular focus that adds bias and reduces its overall usability. As a result, there are
advantages and disadvantages to all UIDLs depending on the particular applications they are used
for [42] [34].

While no UIDL has yet to be fully implemented and widely accepted, URC (Universal Remote
Console) [45] [46] is arguably the UIDL that comes closest to offering a practical, working system.
URC was made an ANSI standard in 2005 [21]. A comparative study of URC with Meshoil—the
XML-based language developed in this research—is discussed in section 6.1 on page 238.

1.5.6 Other areas of research in large-scale control

Ubiquitous computing [44] represents another significant area of research interested in control at
the big picture level. In ubiquitous computing, the focus is on hardware and how to get disparate
devices to interact with one another. The idea is to embed small, networked computers into
everyday objects to provide device interoperation and new means of control (see section 3.8.4 on
page 67). While advances have been made in microelectronics and network technologies in support
of this goal, there still remains the formidable challenge of finding practical ways of achieving
interoperation at the logical level. Current systems are limited in scope and application, often
proving to be difficult to set up and manage without the assistance of specially trained personnel
[8].

Another area of relevance for this research is grid computing [17] [40], where the issue of control
centres not on the user or hardware, but on software. The aim in grid computing is to share
processing loads by having threads of a program run simultaneously on a number of disparate,
geographically distributed devices communicating with each other over a network. Unlike other
models for sharing processing loads such as cluster computing and distributed computing, grid
computing has the advantage that processors do not have to be similar in nature or physically
located close to one another. However, this ability comes at the cost of only being able to support
limited communication and sharing of information between the threads. The Mesh suggests a
way of addressing this limitation by proposing a model for grid computing that is more akin to
concurrency in a conventional programming language seamlessly extended across a network (see
section 3.20.3 on page 109).
1.5.7 Software engineering techniques

The design of the Mesh has relevance to, and is in line with, a number of specific techniques used in software engineering.

To highlight the fact that Meshoil is capable of abstract programming and isn’t just a language for user interface design, another comparative example is discussed in section 6.2 on page 254. The chosen application-specific language for comparison is taken from the completely different discipline of robotics [28]. The example demonstrates Meshoil’s use as a language for supporting embedded device control, with parallels to how such programming languages as Esterel [14] work. Esterel is an imperative, synchronous programming language with a strong bias towards programming control-dominated software and hardware reactive systems.

Meshoil uses a mechanism for code injection that has various parallels in other areas of software engineering (see section 2.1.3 on page 25). One such example is aspect-oriented programming [25], in which the different aspects of a system’s behaviour are individually programmed and then woven together to produce the assembled, final version of the code. Code injection is used in aspect-oriented programming as a convenient technique for simplifying the programmer’s task of writing the program. It does not extend to Meshoil’s ability to support dynamic code insertion in which code is injected on the fly as a program is running. This is the area of dynamically extensible software, an advanced concept in software engineering that finds expression in the design of virtual machines [9].
1.6 Gap statement for this research project

1.6.1 A universally integrated design approach for device control

Much effort has gone into trying to solve the universal usability problem and to come up with a design philosophy that increases usability for all, whatever the type of device being controlled. So far, however, progress has been disappointingly slow.

No practical, universal solution has been found for addressing the ubiquitous computing problem. Current systems are limited in scope and application, often proving to be difficult to set up and manage without the assistance of specially trained personnel—a typical example being Clipsal’s proprietary C-Bus system [8] for controlling devices in buildings via a specially wired network.

These are just two examples of research areas working on the control issue at the big picture level. Each one, however, is limited by a narrow view of the problem and a particular bias. There has yet to be any attempt to address the control issue at the highest level of providing a universally integrated design approach for device control that would achieve the goals of all these research areas.
1.7 The challenge of designing the Mesh

1.7.1 The Web model as starting point

The Mesh would mean a fundamental shift in the way we traditionally think about software design and programming. It is not a case of taking a device programmed in a conventional language and adding some superficial mechanism that allows a measure of Internet connectivity. Rather, the device has to be programmed from the start in a new type of descriptive language that takes its inspiration from the Web model.

The starting point in designing the Mesh is to exploit the strengths of the Web and emulate its main design features.

For example, the Mesh engine is the equivalent of the Web browser. It becomes the universal software application for driving Mesh operation. In the same way that a computer is made Web-compliant by installing a Web browser, a device is made Meshable by installing a Mesh engine. The language of the Mesh is Meshoil, so called because every Mesh engine needs Meshoil to run.

Table 1.2 lists all these basic structural elements of the Mesh and their Web derivatives.

<table>
<thead>
<tr>
<th>Web</th>
<th>Mesh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web browser</td>
<td>Mesh engine</td>
<td>The Web browser is the single, user-level software application that drives the Web. The Mesh engine is the single software application that drives Mesh operation.</td>
</tr>
<tr>
<td>HTML</td>
<td>Meshoil</td>
<td>HTML is the universal language of the Web. Meshoil, the universal language of the Mesh, is a simple, markup language like HTML.</td>
</tr>
<tr>
<td>Web page</td>
<td>Meshoil program</td>
<td>The Web page is the useful end product of the Web. It presents information to the user. A Meshoil program is the useful end product of the Mesh. It describes how a device works and is run by the Mesh engine.</td>
</tr>
<tr>
<td>Web-compliant</td>
<td>Meshable device</td>
<td>A computer with Internet access is made Web-compliant by installing a Web browser. A device with Internet access is made Meshable by installing a Mesh engine.</td>
</tr>
<tr>
<td>computer</td>
<td>Website</td>
<td>A website is the unique location of a Web-compliant computer in the Web. A Meshpoint is the unique location of a Meshable device in the Mesh.</td>
</tr>
<tr>
<td>Website</td>
<td>Meshpointer</td>
<td>A Web page at a website has a unique Web address so that it can be accessed by any Web-compliant computer. A location in a Meshoil program running at a Meshpoint can have a unique Meshpointer to make that part of the program accessible to any Meshable device. Meshpointers use the mesh:// protocol, a hypothetical Internet protocol designed specifically for the Mesh. A full Meshpointer consists of mesh://, the domain name of the Meshpoint and the location within the Meshoil program.</td>
</tr>
</tbody>
</table>

Table 1.2: Emulating the main design features of the Web

The mesh:// protocol would be a request/response protocol between clients and servers much like http://. It would support many of the features of http:// such as file transfer, as well as unique features of the Mesh such as information exchange between running programs. Underpinning it
would be the standard OSI\textsuperscript{5} model for handling information transmission across a network.

1.7.2 The critical factor—design of the Meshoil language

Such differences as the greater degree of interaction complexity that the Mesh must handle create unique challenges in its design. Critical in meeting these challenges is the design of Meshoil—the programming language that underpins the whole Mesh concept.

A conventional language can’t meet the needs of the Mesh, so where to start in the design of Meshoil?

The key to everything is design simplicity.

\textsuperscript{5}OSI stands for Open System Interconnection. It is an ISO (International Standard Organization) standard for worldwide, network-based communication that defines a framework consisting of seven layers. Starting from the top, these layers are called: Application, Presentation, Session, Transport, Network, Datalink and Physical. From the Application layer at the sending location, information is passed down to the bottom layer where it can be physically transmitted. The information then passes back up the layers to reach the Application layer of the receiving location. The Presentation layer translates the application to a standardised network format and vice-versa.
1.8 Open-plan thinking

1.8.1 Searching for a better design philosophy

The Web gave a clue of how to go about Meshoil design by suggesting that the language should be both simple and universal. But the demands placed on Meshoil are much greater than those on HTML. For example, how can a syntactically simple, markup language hope to provide the rich functionality of a conventional language such as Java, let alone go beyond what such languages are capable of doing in supporting a universal means of software interconnectivity?

This section describes how the problem was approached by looking to the science of biomimicry.6

1.8.2 Life is about relationships

The most complex area of design known to us is DNA and the design of life. The complexity of all life forms is expressed in permutations of just four design elements.7 Design information is not represented in the elements themselves, but in the relationships of these elements to one another. A building-brick approach is followed in which the four elements are used in combination to build up a design to any required level of complexity.

Nature shows us that just as it abhors a vacuum, it abhors hard-wired design, since information is encoded in the abstract form of relationships rather than concrete things.

This approach affords the system unlimited descriptive power, since the constraining link between the number of types of design element and the number of designs they can support is broken. It also allows simple tools for handling design to have universal application. The two basic tools for processing biological design information—mitosis for growth and meiosis for reproduction—are identical in all life forms.

1.8.3 Avoiding SIC design

Imagine if, in the role of the creator, we wanted to increase the flexibility of our design options by specifying an organism’s eye colour. Rather than create an eye colour gene as a new sequence of the existing design elements, we invent a fifth element specifically for eye colour. At first glance, this seems a sensible solution. To specify eye colour, all we have to do is to add this element to the organism’s DNA. But now we find that we have to modify the universal growth and reproductive tools so that they can recognise the new element. This makes the tools a little more complicated and harder to use, especially for non-sighted organisms that have no use for an eye colour gene.

If we continue down this path of adding to the list of design elements and making the system more Set In Concrete (SIC), we rapidly find that flexibility in design creation comes at the high cost of making the whole system increasingly unwieldy to use. Instead of enhancing each other, simplicity and flexibility end up as opposing ideals in a process of finding the best trade-off.

DNA isn’t a successful method for describing complex life forms despite its simplicity, but because of its simplicity. This principle seems to be well understood at the base level of the IT industry—there are no calls to improve the digital world by changing from binary to something intrinsically more expressive such as hexadecimal!—but the message gets progressively lost higher up the design scale. Time and time again conventional engineering makes the mistake of trying to solve problems by adding design elements and complicating the design method.

---

6 Biomimicry studies designs in found in nature and imitates them to solve human problems [6]. Such designs are known to work because they have been field tested and proven over time, perfected by evolutionary pressures.
7 The chemical compounds adenine, thymine, cytosine and guanine. Sequences of these compounds form genes that represent particular features of the organism. While IT is based on binary, we can say that nature chooses quaternary as its numeral system since it codes digitally in a choice of four states.
1.8.4 Principles of open-plan thinking

From these observations, three fundamental design principles were extracted. Table 1.3 list these principles, given the collective name by the author of *Open-Plan Thinking (OPT)*.

<table>
<thead>
<tr>
<th>OPT design principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Avoid SIC design.</td>
<td>Minimise the number of hard-wired design elements. Avoid creating design solutions that are SIC.</td>
</tr>
<tr>
<td>2 Encode information in relationships.</td>
<td>Use relationships between the hard-wired design elements to encode design information. Use a building-brick approach in which designs are created by progressively associating larger and larger groups of these design elements.</td>
</tr>
<tr>
<td>3 Make tools simple and universal.</td>
<td>Make tools for processing and handling design information as simple and universal as possible, so that they don’t require constant modification.</td>
</tr>
</tbody>
</table>

Table 1.3: Open-plan thinking

OPT is an argument for using the power of abstraction to solve the problem of design. Much of the advancement of science rests on the spectacular success of the abstraction process. Indeed, it is the only effective tool we have for combating complexity [20]. Applied to design, abstraction means encoding design information in very expressive terms that are able to represent a design to any level of required sophistication. Design is made modular and object-oriented8 (OO).

It’s the Lego principle. Every possible architectural outcome is embodied in the ultimate abstraction of a few, fundamental types of brick, whether you intend to build a ten-brick house or a scale replica of the Eiffel tower. With the right types of brick you can build almost anything.

---

8Object-oriented is commonly used to describe a type of programming that uses abstraction to create models based on the real world. Program structure tends to resemble the system it represents. A modular, building-brick approach is followed in which programs are built up as collections of interacting objects. OO programming utilizes techniques such as inheritance, encapsulation and polymorphism.
1.8. OPEN-PLAN THINKING

The trick is how to craft them so that they can be used in as many ways as possible. The simpler and more expressive they are, the more versatile the solutions they can provide.

Figure 1.4: A building-brick approach
CHAPTER 1. INTRODUCTION

1.9 The concept demonstrator

A concept demonstrator has been developed as an integral part of this research project to:

- demonstrate the feasibility of the Mesh
- provide a platform for demonstrating the main principles of the Mesh in working examples.

1.9.1 Created tools

To demonstrate that the Mesh would not require any special techniques for implementation, the concept demonstrator has been developed using the Internet-friendly Java and XML suite of technologies. XML is a world standard for data exchange that is increasingly being used for computer-to-computer communications. It offers platform independence and a smoother road to ubiquitous computing [19].

In the time available for completing this research project, it has not been possible to develop the concept demonstrator version of the Mesh engine to the point where it can support all the Meshoil language features described in this thesis (unsupported features are clearly marked in the text).

Table 1.4 lists the tools created in support of the concept demonstrator (for details, see Appendix C on page 335).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Written in</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meshoil file</td>
<td>XML</td>
<td>Holds the source file description of a Meshoil program in the form of an XML file.</td>
</tr>
<tr>
<td>Mesh engine</td>
<td>Java</td>
<td>The software application for running a Meshoil program. Reads in, compiles and executes Meshoil files.</td>
</tr>
<tr>
<td>Meshoil tester</td>
<td>XSD</td>
<td>Performs a first-stage validation of the contents of a Meshoil file to ensure that the basic syntax is correct.</td>
</tr>
<tr>
<td>Meshoil viewer</td>
<td>XSL</td>
<td>The stylesheet for converting the contents of a Meshoil file to a graphical format displayable in a Web browser.</td>
</tr>
</tbody>
</table>

Table 1.4: Tools created for the concept demonstrator

The purpose of the concept demonstrator is to demonstrate the functioning aspects of the Mesh, not the appearance or feel of the various tools that might be developed in actual Mesh implementation. The software application representing the Mesh engine, for example, runs as a simple, unadorned GUI that displays the working variables of a Meshoil program. It is not intended to show how the user interface for the Mesh engine might appear to the user in a real example of a Meshable device.

---

9The XML suite of technologies includes XML (eXtensible Markup Language), XSD (XML Schema Definition) and XSL (eXtensible Stylesheet Language). These data management technologies, supported by the World Wide Web Consortium (W3C) [52]), have been designed with the Internet in mind. XML is a meta-language (language shell) for describing markup languages. XSL, written in XML, is a stylesheet language used to translate XML files into other formats such as HTML. XSD, also written in XML, is a validation language for controlling the content and structure of XML files.
1.10 Thesis roadmap

Chapter 2. Getting started with Meshoil (page 23)
Gives a brief introduction to the syntax of Meshoil and discusses a simple, working example of a Meshoil program.

Chapter 3. Principles of the Mesh (page 33)
Discusses the design principles of the Mesh and how Meshoil programs would exploit the potential of the Mesh to deliver unique solutions.

Chapter 4. Universal usability (page 119)
Discusses the relevance of the Mesh to the research area of universal usability.

Chapter 5. Case studies (page 127)
Discusses case studies that highlight the use of Meshoil as a general-purpose programming language for solving a range of conventional programming tasks.

Chapter 6. Comparison to other languages (page 237)
Focuses on the nature of the Meshoil language itself through its ability to emulate the programming constructs of other programming languages in head-to-head working examples.

Chapter 7. Summary (page 269)
Summarises all findings and conclusions of this research work.

Appendix A. Meshoil language description (page 277)
Describes Meshoil syntax and programming constructs.

Appendix B. Discussion on Meshoil (page 323)
Discusses various aspects of the design of the Meshoil language.

Appendix C. Tools (page 335)
Describes the tools used to build the concept demonstrator.
CHAPTER 1. INTRODUCTION
Chapter 2

Getting started with Meshoil

The chapter gives a brief introduction to the syntax of Meshoil—the language of the Mesh—and discusses a simple, working example of a Meshoil program.

The potential of the Mesh is closely bound up in the Meshoil language itself. The aim of this chapter is to give the reader a sufficient knowledge of the language to understand how, in the following chapters, it underpins the principles of the Mesh.

A full description of the Meshoil language is given in Appendix A on page 277.

2.1 Language overview

2.1.1 Design principles

The ultimate measure of a programming language is that it lets you express the things you want to easily and intuitively.

The design of Meshoil is closely based on OPT’s three principles of avoiding SIC design, encoding information in relationships, and keeping tools for processing and handling design information as simple and universal as possible. Specifically, Meshoil is built on the following language design principles:

- an all-encompassing design philosophy
- simple and consistently followed design approach that allows you to guess how a construct will behave from a brief description
- consistent syntax
- small size
- avoidance of exceptional cases and arbitrary limits
- extensibility that allows new constructs to be added easily.
2.1.2 Modular structure

Meshoil is a markup language like HTML but written in XML. Whereas HTML has close to 100 elements, Meshoil, in line with OPT’s principle of keeping the number of hard-wired design elements to an absolute minimum, has just five. These are:

- `<meshoil>` as the root element
- `<a>` for annotation
- `<b>` for brick
- `<c>` for control
- `<d>` for data.

These design elements are the only part of the Meshoil syntax that is SIC. The root element is the single, enclosing element required by every XML file to indicate the nature of its contents. This leaves just four types of element with names like children’s ABC wooden blocks for writing code:

Code is expressed by the contents of these four elements and their relationship to one another. Annotation (`<a>`) is any free text such as a comment. A brick (`<b>`) is the atomic coding unit of a program. A control (`<c>`) is a programming variable which may also represent some software or hardware feature external to the program such as a key on a keypad. Data (`<d>`) is the setting of a control.

Meshoil is a modular programming language. It incorporates many of the principles of OO programming such as abstraction, inheritance, encapsulation and polymorphism (see appendix B.3 on page 329). Code is built up by using the XML elements rather like Lego bricks, with the brick element taking on the significance as the smallest identifiable piece of code. A program is just a collection of brick elements arranged next to each other and nested within each other, each one built up from nested annotation, brick, control and data elements.

Here are six bricks. The two, top-level bricks each contain two nested bricks of their own:

![Figure 2.1: Meshoil’s four XML elements for writing code](image)

![Figure 2.2: Brick nesting](image)
2.1. LANGUAGE OVERVIEW

Each brick can have:

- a *name* to allow the brick to be referenced by name and reused elsewhere in a program (specified in a `<d>` element nested within an `<a>` element for the brick)
- a *namespace*\(^1\) (specified in the `<b>` element’s `namespace` attribute)
- a *spin* to indicate the type of brick and how it is to be processed (specified in the `<b>` element’s `spin` attribute).

2.1.3 Built-in browser functionality through code insertion

A brick can reference another brick by name. This means that any brick can reference remote information, whether on a local server or across the Internet, since *Web browser functionality is built into every brick*.

When a brick name is specified with the Mesh URL prefix of `mesh://`, this referencing takes place over the Internet. Any part of a program written in Meshoil, whether large or small, can interact with any other part of a Meshoil program via the Internet. This interaction can happen at any time: instigated by the user before the program runs, automatically during program loading or automatically during program execution.

Meshoil’s mechanism for code injection has various parallels in other areas of software engineering. One such example is aspect-oriented programming \[25\], in which the different aspects of a system’s behaviour are individually programmed in their most natural form as separate program elements. These program elements are then woven together using a tool called an aspect weaver that produces the assembled, final version of the code ready for compilation and execution. Such a method of code injection is highly specific to how programming constructs are arranged within the language and differs from the more general mechanism provided in Meshoil for inserting code anywhere in a program according to the scheme chosen by the user or programmer.

Code injection in aspect-oriented programming is a technique for simplifying the programmer’s task in writing a program by conveniently compartmentalising the various elements of the task. It does not extend to Meshoil’s ability to support dynamic code insertion in which code is injected on the fly as a program is running. This is the area of dynamically extensible software, an advanced concept in software engineering that finds expression as a useful mechanism in the design of virtual machines where newly generated instructions are directly inserted into the virtual machine’s instruction stream \[9\].

\(^1\)Namespace is a useful naming mechanism for grouping logically related information and increasing the modularity of a system.
CHAPTER 2. GETTING STARTED WITH MESHOIL

2.2 Example of a simple Meshoil program

This section discusses a simple working example of using the Mesh engine to run a Meshoil program. The example is a user interface for controlling a domestic fan, of the type bought at any domestic appliance shop.

2.2.1 Domestic fan

Figure 2.3 on page 27 shows the complete Meshoil program. It consists of a hardware description of the fan followed by a definition of how the user interface controls the running of the hardware. Since Meshoil is written in XML, this program is held in a file called Fan.xml. The program consists of 30 bricks.

The single, top-level brick (<b>) element that contains everything is labelled Fan in an annotation (<a>) element. It contains one control (<c>) element and one data (<d>) element. The control element contains the brick labelled Hardware controls. The data element contains the brick labelled User interface.

The bricks inside Hardware controls define a list of hardware controls that represent the raw, physical capabilities of the device. This list forms an abstract description of every physical feature manufactured into the device. The bricks inside User interface define how the fan works according to the settings of these hardware controls.

The fan has a one-line display, a variable-speed motor, a solenoid for engaging the fan head to make it pan, and two keys for operating the fan. This hardware functionality is represented by the following hardware controls:

1. Display representing the one-line display
2. Motor representing motor speed in the continuous range of 0 (fan off) to 1.0 (maximum speed)
3. Pan representing power to the solenoid in a choice of two states: Off for no panning and On for panning
4. Black key and White key representing the two push-button keys in a choice of two states: Up and Down, with Up being the mono-stable position.
2.2. EXAMPLE OF A SIMPLE MESHOIL PROGRAM

Figure 2.3: Meshoil user interface program for controlling a domestic fan
Figure 2.3 is the graphical representation of the program generated by the Meshoil viewer. Here is what a piece of the XML source code looks like from the Fan.xml file. It is the brick labelled Start fan at the bottom left of the figure that actually starts the fan running:

```
<START>
  <a>Start fan</a>
</START>

<DISPLAY>
  <d>Fan is on</d>
</DISPLAY>

<START>
  <MOTOR>
    <d>Motor</d>
    <d>1</d>
  </MOTOR>
</START>
```

### 2.2.2 Design information encoded in relationships

The associations between the `<c>` and `<d>` elements specified in the various bricks that make up the program define everything about the fan—what it physically consists of and how it works.

For example, the second of the four bricks in the Hardware controls brick defines the Motor control. Embedded in this brick is another control called decimal. decimal is called a spin control because it has a special, recognised meaning rather like a reserved word in a conventional language. It indicates that Motor is to be handled as a decimal-type variable within the program. The brick also contains two nested bricks, the first of which specifies the value of 0.0 for another type of spin control called min. Here, min specifies that the minimum value that the Motor control can take is 0.0.

The User interface part of the program contains bricks that change the values of the Motor and Pan controls. An example of this is the brick that sets the motor control (`<c>` element) to 1 (`<d>` element) for full speed running as part of the procedure labelled Start fan.

The fundamental association between a `<c>` and `<d>` element is repeated at the higher level of the Hardware controls and User interface bricks themselves. The Hardware controls brick is given ‘with’ spin and the User interface brick is given ‘do’ spin to identify these two bricks as the equivalent of the data and executable part of a conventional language. For a user interface, they provide the means to say: ‘With these hardware features, do these things’. Instead of defining a single assignment, they effectively define the group of all the assignments to be made for the hardware controls. Figure 2.3 shows how the Meshoil viewer displays the spin given to a brick in parentheses immediately above the brick.

### 2.2.3 Hardware controls

The manufacturer of the fan is responsible for defining the Hardware controls brick and for supporting its physical connection to the device hardware. This brick forms an abstract hardware layer between the device and the user interface.

The direction of data flow between the Meshoil program and external world is specified by the in, out and inout spin controls. For example, in spin for the Black key and White key controls indicates that when these keys are physically pressed, the Meshoil program will be able to detect
2.2. EXAMPLE OF A SIMPLE MESHOIL PROGRAM

the event and react accordingly by checking whether the settings for these controls have changed from Up to Down. However, no hardware mechanism is supported for allowing the program to move the keys up or down.

The reverse is the case for the Display control that has out spin. The program can change what is physically displaying on the fan’s display screen at any time by simply changing the setting of the display control.

2.2.4 User interface design

The Black key is used to control fan running and the White key is used to control panning. The user interface is represented by two bricks that specify how each key works.

Use of the if spin control at the top of each of these bricks makes the execution of the bricks they contain conditional on the brick’s evaluation at runtime. Adding if spin changes a brick from acting like an assignment statement to acting like an IF statement in a conventional language. By default, nested bricks execute when the parental brick condition is true. But if a brick is given not spin, it executes when the parental brick condition is false.

In the Black key operation brick, changes to the settings of the Display, Motor and Pan controls are made when the Black key is Down. In the White key operation brick, changes to the settings of the Display and Pan controls are made when the White key is Down.

2.2.5 Making the keys work correctly

If the keys were bi-stable, it would be possible to relate fan running and panning directly to the state of the two keys. For example, Up for the Black key would mean fan off (Motor setting 0), while Down for the Black key would mean fan on (Motor setting 1). But the keys are mono-stable, push-button type. The user can’t be expected to hold them down continuously to keep the fan running or panning! The answer is to make the keys toggle fan running or panning each time they are pressed.

The Black key toggles fan running by setting the Motor control to 1 or 0, depending on the current setting of this control when the key is pressed. While the fan is running, the White key does the same toggling operation for the Pan control.

One further refinement is necessary to make the user interface work correctly. Starting or stopping fan running or panning must only happen once each time a key is pressed. It should make no difference how long the user holds the key down. A start or stop procedure should only execute once, and not repeatedly execute in an alternating pattern for as long as the key remains held down. This is achieved by including the rise spin control in the specification of how each of the keys work in the Black key operation and White key operation bricks. Using rise ensures that nested bricks only execute at the instantaneous moment that the parental condition becomes true.
2.2.6 Running the fan user interface

Figure 2.4 shows a sequenced example of the running the fan user interface.

Frame 1  Running the concept demonstrator brings up windows that display the controls used within the Meshoil program (see appendix C.1 on page 335). These windows provide a complete snapshot of the current state of the running program. This window shows the state of the fan’s five hardware controls immediately after the Mesh engine has loaded and started executing the Meshoil program. The fan is initially off, represented by the Motor control setting of 0.

Frame 2  The user ‘presses’ the Black key to start the fan running. In the simulation, this is done by using the mouse in the window to display the two available settings for the Black key control and selecting the Down setting (the figure shows how this temporarily masks the setting for the White key control).

Frame 3  As soon as the Black key is ‘pressed’, the fan starts running, as indicated by the message showing on the Display and the value of Motor changing from 0 to 1.

Figure 2.4: Running the fan user interface (1 of 3)
2.2. EXAMPLE OF A SIMPLE MESHOIL PROGRAM

Frame 4  A fraction of a second after ‘pressing’ the Black key, the key returns to its Up position ready to be ‘pressed’ again. In the real world, the fan’s two mono-stable keys would immediately return to their Up position automatically. In this simulation of user interface operation, however, the effect has to be simulated in Meshoil. The necessary code (not shown in figure 2.3 on page 27) is included in the Fan.xml file holding the fan program and runs as an independent thread in parallel with the fan’s user interface.

Frame 5  The user ‘presses’ the White key to start the fan panning.

Frame 6  As soon as the White key is ‘pressed’, panning starts as indicated by the message showing on the Display and the value of Pan changing from Off to On.

Figure 2.4: Running the fan user interface (2 of 3)
Frame 7  The White key immediately returns to its Up position ready to be ‘pressed’ again.

Figure 2.4: Running the fan user interface (3 of 3)
Chapter 3

Principles of the Mesh

This chapter uses a working example of a Meshoil program to demonstrate the following principles of the Mesh:

- **Building-brick design flexibility** (page 36)—building up any level of required program functionality by mixing and matching discrete blocks of code
- **Atomic-level networking** (page 36)—making physically discrete, concurrently executing programs interact in any way that is desired
- **Abstract hardware layer** (page 37)—the ability to control hardware without any knowledge of how it physically works
- **Cost-free user flexibility** (page 38)—the benefits of a flexible design approach with none of the traditional drawbacks
- **Universal usability** (page 41)—an original contribution for solving this problem
- **Futures programming** (page 46)—writing programs that anticipate the future by controlling hardware that has yet to exist
- **Transparent programming environment** (page 48)—making housekeeping tasks traditionally associated with programming transparent for the programmer
- **Virtual logic** (page 58)—changing the logic of how a program runs without editing code
- **User-generated software** (page 60)—Web 2.0-style software created directly by the user
- **Virtual hardware** (page 64)—adding hardware functionality to a device without physically installing anything
- **Ubiquitous computing** (page 67)—a simple, practical mechanism that supports computers being increasingly interwoven into the fabric of our lives
- **Responsibility optimisation** (page 68)—redistributing the responsibilities of users, programmers and manufacturers in such a way to minimise the overall workload and achieve a better solution for all
- **Ghost-in-the-machine API** (page 73)—instantaneously upgrading an electronic device, no matter how basic, to the level of a programming platform
CHAPTER 3. PRINCIPLES OF THE MESH

- Abstract API layer (page 74)—maintaining platform-independent programming even when making use of a platform’s API
- Engineering integration by layering (page 74)—the ability of a Meshoil program to fit into the bigger engineering picture as self-contained, easy-to-connect functional layer
- Software-team scalability (page 77)—scalability that isn’t just a feature of the language but is represented in those using the language too
- Cut and paste programming (page 78)—robust support of code manipulation and the freedom to paste code almost anywhere in a program
- Globally-unified hardware control (page 80)—standardising how hardware is controlled in the real world
- Program morphing (page 84)—the process by which a program changes into a different program while running
- Hardware reuse (page 86)—reusing the existing hardware features of a device to provide the user with what appears to be a different device
- Interface agents (page 89)—making autonomous decisions designed to anticipate users’ needs and make life easier
- Universal agent (page 90)—replacing the need to program interface agents individually by using a universal agent capable of autonomously customising the operation of any type of device
- Superprogramming (page 93)—unlimited flexibility in device interoperation across the Mesh
- Point-accurate interoperation (page 97)—achieving maximum flexibility of device interoperation with minimum effort by modifying code at the exact location where interoperation is designed to have its effect
- Grid computing (page 109)—supporting the concept of program threads running simultaneously on geographically distributed and disparate processors
- Ghost processing (page 109)—boosting the computing power of a device through the use of virtual processors
- Code filtering (page 115)—filtering out errors from a program without having to edit the code directly
- Code integrity through accreditation (page 116)—using a strategic approach to manage the problem of unreliable and malicious software.
3.1 A house with two fans

This chapter builds on the simple example of a user interface for a domestic fan described in section 2.2 on page 26 to highlight the various design principles on which the Mesh is based.

To simulate the behaviour of devices connected via the Mesh, two programs are effectively run as one. The two program halves run independently of each other as separate threads, representing the user interfaces of two fans that each have an Internet connection to the Mesh.

3.1.1 Running this simulation

Figure 3.1 shows the main control window during a program run.

![Main control window for simulating the operation of two fans](image)
CHAPTER 3. PRINCIPLES OF THE MESH

The fans are called fan1 and fan2. The top two groupings, each with five controls, display the hardware controls for the two fans. Underneath this are two groupings of controls for setting up how the simulation runs for each fan. Towards the bottom of the window is a single control that represents temperature information made available on the Mesh by the Meshpoint of the Weather Bureau. Right at the bottom of the window is the Meshpoint of a personal computer attached to the Mesh.

The simulation can be run with a choice of user interfaces for each of the two fans. For example, the control called fan1 USER INTERFACE controls which user interface is currently in use for fan1. This control can be seen at the top of the third grouping of controls in the control window. The figure currently shows that the Standard (1-speed, pan) user interface is selected for fan1, while the Advanced (3-speed, pan) user interface is selected for fan2. fan2 is also running four software plugins for its currently selected user interface: Convert 3-speed to 5-speed plugin, Thermostat plugin, Panning timeout plugin and Remote control from other fan plugin.

All Meshoil used to run this simulation is held in a single file called 2Fans.xml.

3.1.2 Building-brick design flexibility

Mesh support for the principle of Building-brick design flexibility is demonstrated throughout the examples that follow in this chapter for how the fans can be made to operate.

Each example uses a different version of the user interface program in order to highlight a particular set of Mesh principles. These program versions are assembled from a collection of code modules, some embedded in others, that show how it is possible in Meshoil to build up any level of required program functionality by simply mixing and matching discrete blocks of code.

3.1.3 Atomic-level networking

A language built on the building-brick approach means that networking capability can be wired into a program at the very lowest level—the atomic unit of code. Such atomic-level networking means that physically discrete, concurrently executing programs can be made to interact in any way that is desired. This is demonstrated in the examples towards the end of this chapter that discuss ways of making the two fans interoperate.
3.1.4 Abstract hardware layer

The abstract hardware layer allows a Meshoil program to control hardware without any knowledge of how the hardware physically works. Every feature of the hardware is made accessible to the program in abstract form as a simple programming variable.

Figure 3.2 shows the hardware controls making up the abstract hardware layer of fan1. The list of controls is unchanged from the previous fan example, discussed in section 2.2.3 on page 28, except that here a namespace is used to give the controls a unique Mesh identity. To make the names compatible with naming rules for Internet addressing, they are written in lowercase without embedded spaces.

The names of the five hardware controls—display, motor, pan blackkey and whitekey—are automatically prefixed with the mesh://fan1 namespace since this namespace is declared at the top of the brick and automatically passes down to all contents of the brick by inheritance. (In a real example of referencing code across the Internet, the namespace would be declared as fan1, not mesh://fan1, since specifying the mesh:// prefix would only be required client-side. In this simulation, however, client-side and server-side naming must correspond exactly since all cross referencing occurs within a single XML file.)
CHAPTER 3. PRINCIPLES OF THE MESH

3.2 “I don’t want to change anything”

Control settings for this simulation run (same as default settings when program execution begins):

```
- fan1 USER INTERFACE
  - -- Current 1-speed to 5-speed
  - -- 1. Blackkey on/off
  - -- 2. White key 5-speed
  - -- 3. Black and white key panning
  - -- 4. Thermostat
  - -- 5. Panning timeout
  - -- 6. Safe (slow)
  - -- Remote control from other fan
  - -- Close of other fan operation
  - -- Only usable if other fan is on
  - -- Start panning if other fan stops panning
```

Figure 3.3: Simulation settings: Standard (1-speed, pan) user interface

3.2.1 Standard (1-speed, pan) user interface

Figure 3.4 on page 39 shows the Standard (1-speed, pan) user interface. It is the same as the user interface from the previous fan example—the blackkey toggles the fan on and off, and the whitekey toggles panning while the fan is running.

There are two differences, however. Firstly, all controls now belong to the fan1 namespace. Secondly, key operation has been made more precise so that a key only works when it is the only key being pressed. For example, a blackkey press will no longer do anything if the whitekey is held down at the same time. This refinement is needed to ensure that the keys work correctly in the examples that follow where shift key operations are added to the user interface.

In the real world, the code represented by the Standard (1-speed, pan) user interface brick would simply be loaded in the fan’s Mesh engine to run unconditionally, but for the purpose of being able to switch user interfaces in this simulation, the standard user interface only executes when fan1 USER INTERFACE is set to Standard (1-speed, pan).

3.2.2 Cost-free user flexibility

A common misapprehension about any system that offers considerable design flexibility is that it requires a lot more effort from the user in getting everything set up and working in the first place. This is not the case with Meshoil. Any Meshable device can initially be set up to work a certain way, for example by the device manufacturer, without ever subsequently needing to be changed or modified. For all intents and purposes, the device can appear to the user just as if it were programmed to work as if it had been coded in a conventional language.

In this first example, the fan is sold to the customer with the standard user interface already installed. The fan works the first time he plugs it in without any fuss. He is quite happy with how it works and never wants to change anything.

This begs the question: when flexibility isn’t needed, are there disadvantages in using a flexible system instead of a dedicated and inflexible one? The answer is only yes if the flexible system creates penalties or additional overheads for the user, such as making the device harder to use or degrading its performance in some way.
3.2. "I DON'T WANT TO CHANGE ANYTHING"

Figure 3.4: Standard (1-speed, pan) user interface (1 of 2)
The well-known children’s Lego brick is a good example of this. Lego bricks are easy to manufacture and use, but a particular design built in Lego never looks as authentic as its custom-built equivalent. There is a small trade-off to be made in using such an abstract design system, despite the many advantages it offers. If you are only ever going to build the same design and never change it, you would be better off not using Lego at all and using building materials specifically suited to what it is you are making instead.

But this is not the case with Meshoil. As this example shows, Meshoil embodies cost-free user flexibility because the user doesn’t have to pay a price for having the flexibility to change how the fan works, even when he has no need to make use of this flexibility. Empowering the user with greater freedom in creating solutions hasn’t led to a trade-off in everyday usability. The Meshable fan works just the same as if it had been programmed using a conventional language—a distinction that he can choose to remain blissfully unaware of.
3.3 “I want something simpler to use”

Control settings for this simulation run:

```
+ fan1 USER INTERFACE
  + -- Convert 3-speed to 5-speed
  + -- 1. Black key on/off
  + -- 2. White key 5-speed
  + -- 3. Black and white key panning
  + -- 4. Thermostat
  + -- 5. Panning timer
  + -- 6. Safety cutout
  + -- Remote control from other fan
  + -- Clone of other fan operation
  + -- Only usable if other fan is on
  + -- Start panning if other fan stops panning

Simplified (1-speed)
```

Figure 3.5: Simulation settings: Simplified (1-speed) user interface

3.3.1 Simplified (1-speed) user interface

What if the user isn’t completely happy with how the fan works after he takes it out the box? A common experience suffered by many is a user interface that is difficult to use because it tries to do too much. The device is full of features that the user does not want. These unused features are worse than useless because they clutter up the user interface making everything harder to use.

This is demonstrated here by our fan user who decides that he doesn’t like the panning feature and never wants to make use of it. Instead, he would prefer the convenience of being able to switch the fan on or off by pressing either of the two keys, without having to remember which is the one that works. Figure 3.6 on page 42 shows the simplified version of the user interface that achieves this.

To simulate that the simplified user interface is downloaded from a remote location to replace the existing user interface, the new code is not directly located in the software for fan1, as was the case for the standard user interface shown in figure 3.4 on page 39. Instead, it is inserted from another location within the 2Fans.xml file by referencing its brick name of mesh://ui4u/fan.interface.simplified (1-speed). Figure 3.6 shows how the Meshoil viewer displays the name of a brick to be inserted immediately above the inserted brick with that name. All contents of the inserted brick display faint, including its underlined brick name.

3.3.2 Universal usability

Meshoil’s approach of abstracting the hardware of a device in the abstract hardware layer maximises the uses that the device can be put to and provides one of the pillars for solving the universal usability problem (discussed in greater detail in Chapter 4 on page 119).

The with brick shown in figure 3.2 on page 37 is a complete description of everything that the device is fundamentally capable of doing. The user interface code shown here in figure 3.6 describes a particular way of exploiting the device’s hardware. The task of changing how the fan operates is simply a matter of leaving the hardware description in place and plugging in a new block of user interface code that references the hardware description.

Conventional languages do not provide this clear layer of separation between what a device is physically capable of doing and what it is actually used to do. They don’t have the means to interface so easily with device hardware. As a result, it is hard to change how the device works. While there is no technical reason why user interfaces written in conventional languages cannot be switched like this, it isn’t a practical solution since the job demands an intimate knowledge
CHAPTER 3. PRINCIPLES OF THE MESH

of the particular hardware design of the device that only a few, highly skilled and specialised programmers would have. Attempts have been made to provide software modules to do this, but none have proved practical as a universal method for user interface design [5] [18].

Figure 3.6: Simplified (1-speed) user interface
3.3.3 The conventional design trade-off

Not being able to change how a device works once the user interface has been written creates headaches for the software designer and user alike. The problem is well understood, but not the reason for it or the cure (Figure 3.7).

![Figure 3.7: The conventional trade-off](image)

Conventional wisdom believes that flexibility in user interface design, far from being the beneficial consequence of a simple approach, is the very reason why everything gets so complicated. Flexibility is interpreted to mean the superficial process of increasing user options by packing more user procedures into the user interface. Design is seen as the inevitable process of seeking the best trade-off between conflicting needs for simplicity and flexibility [33]. Simple means easy to use, but few features. Flexible means many features, but difficult to use.

This dilemma is well documented in the literature. Vanderheiden [50] comments on the dangers of the conventional approach of trying to increase flexibility and usability by simply adding more and more features. He notes that designers tend to get overwhelmed in the process and add the easiest ones to implement that are not necessarily the most useful—the problem of ‘low hanging fruit’. Baumann [4] sees feature overflow—bloatware—as the biggest challenge in user interface design since every additional feature increases the complexity of the system. Norman [32] estimates that this cost in complexity grows in proportion to the square of the number of features. Newell [31] argues that trying to follow the principles of universal design places such unreasonable demands on designers that it tends to inhibit them from attacking the problem at all. He believes that making products flexible enough to suit all types of user including the disabled can make them significantly more difficult to use for those without disabilities.

An often stated fear is that designing for all types of user inevitably leads to ‘dumbing down’. By accommodating the low-skilled user, we end up with a lowest common denominator design that fails to satisfy the majority of users.
CHAPTER 3. PRINCIPLES OF THE MESH

3.3.4 The software designer’s current headache

A simple, concrete example illustrates why the current approach to software design forces us to choose between simplicity and flexibility. One of the special features of Ericsson’s R600 mobile phone is its ability to allow the user to link specific callers to a particular background colour of the screen. You can immediately tell from a glance at the screen who is calling.

There is nothing wrong with this user feature, but why go to all the trouble of creating a multi-coloured screen and then make such little use of it? Why limit it to indicating callers? Why can’t it be used for many other things that the user might prefer, such as indicating when the battery is low, a particular user profile is active or call waiting is off (Figure 3.8)?

![The software designer’s current headache](image)

Figure 3.8: The software designer’s current headache

All the many other things that the user might want to do with the multi-coloured screen have been programmed out of the user solution space. It seems that Ericsson’s design team did all the hard work in building the feature into the handset, only to fall short at the last hurdle by denying the user proper access to it.

But what choice did Ericsson have? Knowing that whatever solution they provided couldn’t be changed once the phone left the factory, all they could do was to second-guess what features users might want and program the user interface accordingly, hoping to get the balance right. Inevitably, the result will be some users frustrated by too few features and others frustrated by too many.

Every unused feature in a user interface increases clutter and detracts from the overall usability of the user interface. This is why it would have been no solution to try satisfying everyone by providing user procedures for linking screen colour to every feature in the phone. No user would be likely to want to use them all, and all that would be achieved is a user interface that would annoy everyone. And even if this were possible, the result would still be a subset of what the device is innately capable of doing with its hardware, since there would still be an uncountable number of unsupported ways that these features could be used in combination with others to achieve specific outcomes for the user (eg, instead of displaying a red screen all the time the battery is low, display it just as a brief reminder when there is an incoming call).
3.3.5 Expanding the user solution space

What is really needed is to be able to customise the design of the user interface for each user—to provide the user with all the features they want and none of the ones they don’t. The focus should be on increasing the solution space available to the user.

For example, if a user is only interested in using screen colour for indicating which profile is active, then they should be provided with a user interface that supports this function and no more. Such an approach to user interface design is impractical using conventional programming methods, since there is no easy way to tap into the hardware capability of the device in the way that Meshoil can with its abstract hardware layer.

Ultimate solution space is set by device hardware, since it is this that determines what the device is physically capable of doing. Clearly, a device cannot do more than what it has been engineered to do—you can’t use a mobile phone to boil an egg. Meshoil’s abstract hardware layer preserves this ultimate solution space. In the Meshoil equivalent of the Ericsson phone, providing a colour selectable screen would be represented by the addition of a hardware control called Screen colour that had a choice of Red, Green or Blue settings. The user interface for the phone would then be free to exploit this hardware feature in any way it liked (Figure 3.9).

We can sum up the philosophical difference in the design approach like this: the manufacturer can afford the luxury of programming the user interface to work in exactly the way the user might like, because they know that if they get it wrong, the user can always throw the user interface out and get a new one. Compare this with the conventional design approach: a user interface is for life, so to cover all bets the manufacturer packs in as many user features they think they can get away with without making the user interface too unwieldy.

![Expanded user solution space](image-url)
3.4 “I want better indication of what the fan is doing”

Control settings for this simulation run:

![Simulation settings: Simplified (1-speed, pan, LED) user interface]

3.4.1 Standard (1-speed, pan, LED) user interface

Our fan user changes his mind again and decides he likes the way the standard user interface works after all. But surfing the Mesh one day he comes across a version of the standard user interface that supports the additional feature of an LED for indicating when the fan is running. As in the case of the simplified user interface, he has only to download and install this new user interface. He finds that the fan works exactly the same way as before, but of course no LED lights when the fan is running since the fan doesn’t have one.

Figure 3.11 on page 47 shows the LED version of the standard user interface. The designer of this user interface could simply have programmed the blackkey to toggle both the fan and the LED on and off, but instead chose to make the LED an integral part of how the fan runs. The blackkey toggles the LED on and off, and toggling the fan on and off is made dependent on the LED switching on and off.

3.4.2 Futures programming

How does the fan still manage to work when its operation now depends on a item of hardware that does not exist? Running the user interface does not cause any problems or generate any ‘unsatisfied reference’ errors because the Mesh engine, not finding any definition of this LED anywhere in the program, automatically creates LED as a local programming variable. The setting for this programming variable changes as if it was switching an LED on and off, but to no resultant effect in the hardware.

All controls within the executable part of Meshoil program (code within a do brick) are treated as simple programming variables, regardless of any external associations they may have. No distinction is made between a control such as blackkey, that is an abstract representation of a item of hardware, and a control such as LED that has no identity outside the program.

If our fan user subsequently requested the supplier of the fan to upgrade the unit by installing an LED, the supplier would physically mount the LED on the fan and represent this in Meshoil by adding LED to the with brick that lists all the fan’s hardware controls. The user interface, and any other code that makes reference to a control called LED, would then operate the LED as intended.
3.4. “I WANT BETTER INDICATION OF WHAT THE FAN IS DOING”

Figure 3.11: Standard (1-speed, pan, LED) user interface (1 of 2)
CHAPTER 3. PRINCIPLES OF THE MESH

This example demonstrates how Mesoil supports the concept of *futures programming*—the writing of programs that anticipate the future by controlling hardware that has yet to exist. In the same way that a financial commodity can be traded in a futures exchange based on the future condition and value of that commodity, a user interface can be written in Mesoil based on the future capability of the device that it controls. Code can be added to the user interface to control hardware that does not yet physically exist in the device. If at some future point this hardware is installed in the device, it will instantly work without any software modifications needing to be made.

### 3.4.3 Transparent programming environment

By following OPT’s principle of making tools simple and universal, the Mesh is able to provide a *transparent programming environment*. Mesoil makes the task of programming simpler by automating housekeeping tasks traditionally associated with programming, making them transparent to the programmer.
In the example here, the Meshoil programmer doesn’t need to be aware of which controls are just simple programming variables and which have greater significance in representing parts of the hardware. The programmer is able to integrate the functioning of device hardware directly into the body of a program without having to use any programming constructs for identifying the controls that have this special function, since the semantic linkage is conveniently performed elsewhere.

The example of the LED control also shows how a control can be used in a program without any prior declaration, since in the absence of an explicit definition the characteristics of a control, such as its datatype, will automatically be determined at compilation time according to how the control is used in the program (see Appendix A.4 on page 302).

No special mechanisms are needed for multithreading a program, since concurrency and the definition of how information is to be shared between multiple threads is determined from how the blocks of code are arranged in the program (see appendix A.4.4 on page 304).
3.5 “I want the fan to do more”

Control settings for this simulation run:

- Fan USER INTERFACE
- → Convert 3-speed to 5-speed
- → 1. Black key on/off
- → 2. White key 5-speed
- → 3. Black and white key panning
- → 4. Thermostat
- → 5. Pan/Throttle
- → 6. Safety output
- → Remote control from other fan
- → Clear other fan operation
- → Only usable if other fan is on
- → Start panning if other fan stops panning

Figure 3.12: Simulation settings: Advanced (3-speed, pan) user interface

3.5.1 Advanced (3-speed, pan) user interface

This time, our user wants to do more than what the standard user interface offers. He wants to be able to select different fan speeds rather than just having the fan always on at full speed.

The reason why this feature wasn’t offered in the standard user interface is that the fan only has two keys, giving it limited means of physical control. Adding variable speed as an additional user feature means having to introduce the concept of shift key operations, something that the manufacturer decided most users wouldn’t like.

Figure 3.13 on page 51 shows this advanced user interface. Our user is prepared to put up with the fan being a little harder to operate for the gain in functionality. Unlike the conventional software design trade-off discussed earlier, there is no way of escaping this trade-off because we are up against the ultimate limit to flexibility—the design of the hardware itself.

The whitekey is now used to toggle a choice of three speed settings: slow, medium and max, corresponding to settings for the motor control of 0.3, 0.7 and 1.0. The whitekey is also used as a shift key with the blackkey to toggle panning on and off.
3.5. “I WANT THE FAN TO DO MORE”

Figure 3.13: Advanced (3-speed, pan) user interface (1 of 3)
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.13: Advanced (3-speed, pan) user interface (2 of 3)
3.5. “I WANT THE FAN TO DO MORE”

Figure 3.13: Advanced (3-speed, pan) user interface (3 of 3)
CHAPTER 3. PRINCIPLES OF THE MESH

3.6 “I want to modify an existing feature”

Control settings for this simulation run:

```
- fan1 USER INTERFACE
  - -- Convert 3-speed to 5-speed
    - -- 1. Black key on/off
    - -- 2. White key 5-speed
    - -- 3. Black and white key to 5-speed
    - -- 4. Thermostat
    - -- 5. Panning timeout
    - -- 6. Safety cutoff
    - -- Remote control from other fan
    - -- Only usable if other fan on
    - -- Start panning if other fan stops panning
```

Advanced 5-speed, pan
Downloaded

Figure 3.14: Simulation settings: Convert 3-speed to 5-speed plugin

3.6.1 Convert 3-speed to 5-speed plugin

Our user finds he likes the variable speed feature, but would prefer a choice of five rather than three speeds. One way to achieve this is to download a different version of the advanced user interface from the Mesh that offers 5-speed instead of 3-speed control. In the simulation, this can be done by setting the fan1 USER INTERFACE control to the Advanced (5-speed, pan) setting.

But there is an even simpler solution that does not involve replacing the whole user interface. All the user needs to do is to download the software plugin called Convert 3-speed to 5-speed plugin, shown in Figure 3.15 on page 55, and install it in the fan by placing it alongside the code of the existing user interface. He doesn’t need to do anything more than this.

The existing user interface code remains untouched and continues to run as before, but somehow 3-speed selection has disappeared, replaced by the new 5-speed selection.

3.6.2 Preserving existing parts of a design

Adding a software plugin rather than replacing a complete program is useful in cases where existing software needs to be preserved and not overwritten by importing new code. In user interface applications, for example, the user interface may have been personalised and the user may not want to lose the changes by overwriting the existing user interface with a completely new one.

Here, the downloaded plugin represents just that part of the user interface responsible for providing a choice of five speeds. It has the effect of replacing the section of code inside the existing user interface responsible for speed selection. Pressing the white key now toggles between the five speed settings 1–2–3–4–max instead of slow–medium–max.

The code for providing the choice of three speeds is still part of the user interface. It has not been physically edited out or changed in any way, so how has it managed to be switched off so that it does not interfere with the new code that provides the five speeds?
3.6. “I WANT TO MODIFY AN EXISTING FEATURE”

Figure 3.15: Convert 3-speed to 5-speed plugin (1 of 4)
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.15: Convert 3-speed to 5-speed plugin (2 of 4)
Figure 3.15: Convert 3-speed to 5-speed plugin (3 of 4)
3.6.3 Virtual logic

In a conventional language, code has to be written at the physical location in the program where it has its effect. To change how the program runs at a certain location, you naturally edit the code there. In Meshoil, this SIC restriction on programming methodology is relaxed. Code can be written elsewhere, yet have the same effect as if it had been physically inserted at the intended location—virtual logic is made possible. When the program runs, the new code acts like an overlay, modifying how the original code executes.

Virtual logic offers a number of advantages. For the programmer, it gives greater freedom in controlling program structure. It means you can change how a program works without making any physical changes to it. Instead of having to edit code at many locations throughout a program, the code representing all the changes can be written at a single, convenient location. This simplifies the process of implementing change. Should the modifications need to be reversed at any time, the task is a simple matter of removing the overlay as a single, easily identified block of code. There is no longer any need to pick through the program to find all the places where changes were made in order to edit them out.

For the end user, virtual logic means easier customisation. For example, a program might contain a number of such overlays, each one modifying the main body of the program a different way. The program would effectively contain the means for its own editing, with the user able to customise software at any time by switching overlays on and off, either when starting the program or dynamically during runtime.

Virtual logic is made possible by the flexible way that programming scope is defined in Meshoil. In a conventional language, the scope of code is always tightly bound. Scope can be bound in this way in Meshoil (see appendix A.4.4 on page 304), but there are times such as this one when a looser scope has its advantages.

In this example, all user interface code remains in global scope because it is defined by a single do brick. This means that the whole program executes in one large, repeatedly executing loop. The plugin is positioned at the same level as the brick that defines the user interface. By setting up this parallel alignment and by mimicking the logical pathways of the user interface in the plugin, conditions can be set up in the plugin with the same effect as if they had been directly specified in the main body of the user interface.
3.6.4 Path hijack and release

The plugin code interacts with the code in the main body of the user interface by the process of path hijack and release. Code in the plugin hijacks the flow of logic so that the equivalent code in the user interface no longer executes, effectively switching off the unwanted code.

In the example here, the first point of hijack occurs when the blackkey is used to switch the fan on at the slow speed (motor setting 0.3). This setting is changed to the new minimum speed setting of 1 (motor setting 0.2). This disables the speed control code in the main body of the user interface, since it is not programmed to recognise 0.2 for the motor control. Each successive toggling of the speed by the plugin continues to be unrecognised until the speed reaches the maximum setting of 1.0, the point at which the path is released since the main body of the user interface recognises this value and switches on again.

In this simple example, the plugin immediately hijacks the path again to restart the cycle, since the intention is for the plugin code to completely replace the equivalent code in the user interface. But in many situations this would not be the case. The intention might be to augment or modify existing procedures without replacing them.

Examples of these more subtle interactions are given in section 5.7 on page 217 that describes the user interface for a Nokia 6210 mobile phone. A plugin provides everything required to add Nokia’s profiles feature to the user interface—involving changes to the existing phone book, menu structure and incoming call procedures. Despite the fact that profiles are closely interwoven with the main features of the user interface, virtual logic makes it possible to write all the associated code at a single location as a self-contained overlay. Running the Meshoil program in the concept demonstrator shows how the Nokia user interface can be instantly transformed to support every aspect of the integrated profiles feature by simply switching this overlay on.
CHAPTER 3. PRINCIPLES OF THE MESH

3.7 “I want to assemble my own plug-and-play design”

Control settings for this simulation run:

```
- fan1 USER INTERFACE
- -- Convert 3-speed to 5-speed
- -- 1. Black key on/off
- -- 2. White key 5-speed
- -- 3. Black and white key panning
- -- 4. Thermal dot
- -- 5. Panning timeout
- -- 6. Safety control
- -- Remote control from other fan
- -- Only usable if other fan is on
- -- Start panning if other fan stops panning
```

Figure 3.16: Simulation settings: DIY user interface—Web 2.0 style

The settings shown above simulate the effect of not having a user interface per se (fan1 USER INTERFACE control set to '-') and instead, downloading software plugins for various individual user interface features.

3.7.1 User-generated software

Having found how easy it is to download different user interfaces for his fan, our user becomes more adventurous and decides to assemble a user interface of his own. Taking his cue from the Web 2.0 model for user-generated content, he chooses to create user-generated software. No longer seeing himself a passive consumer, he feels empowered to generate solutions of his own. After all, why rely on professionals when you can do just as good a job yourself? Why settle for a one-size-fits-all solution when you can get exactly what you want? The Mesh provides him with the means to be creative in an area that was completely unintelligible to him before: the arcane world of computer programming. Programming how something is to work has become a tool within everyone’s reach.

He doesn’t actually require any programming skills, since all he is going to do is download self-contained software plugins that will provide the various user features he wants. Building up a user interface this way is rather like making something from a DIY kit. All the hard, detailed work is already done for you. You just do the final assembly.

Starting from scratch, he downloads the first of three plugins that provide the equivalent functionality of the Advanced (5-speed, pan) user interface: the plugin for using the blackkey to toggle the fan on and off, the plugin for using the whitekey to select one of five speed settings, and the plugin for using the blackkey and whitekey together to toggle panning on and off.
Figure 3.17 shows the first of these plugins.

3.7.2 User Interfaces For You Meshpoint

The user found these plugins at the same Meshpoint where he found all the user interface software described in the earlier sections. This Meshpoint, with the Meshpointer ui4u (User Interfaces For You), is a Meshoil shareware site specialising in user interface software for domestic appliances. It serves as a meeting ground for users who are looking for more interesting, relevant and personalised ways of running their home appliances.
Figure 3.18 shows this Meshpoint, simulated in this working example by being just another location holding code in the 2Fan.xml file.

![Image of Meshpoint](image.png)

**Figure 3.18: User Interfaces For You Meshpoint**

### 3.7.3 Software as a tradable commodity

User-generated software means that software can be treated as a commodity, something that is easily traded. Software can now be bought, sold, distributed, shared, exchanged and licensed like any other commercial product. This trend for the commodification of content is already evident in the Web, where websites such as YouTube [54] allow users to upload content to be made available to other website users.

YouTube have policies that cover issues like the nature of the content and the intellectual property rights of those who upload content. Owners of content have a responsibility not to submit material that is copyrighted, or is unacceptable through being offensive, defamatory or misleading. Content owners retain ownership rights but grant YouTube a worldwide, non-exclusive, royalty-free license to display their works on the website.

The Mesh would build on concepts like these for managing software. In the example here, the ui4u Meshpoint might charge users for downloading software and have an arrangement for sharing proceeds with software providers who upload software. Issues arising from these new styles of management, such as the problem of unreliable and malicious software (discussed at the end of this chapter), would need to be addressed.
3.7.4 Too much freedom?

The Mesh makes it easy to change how a device is programmed to work, but would manufacturers like the thought of having their products reprogrammed so easily? Would a company like Nokia, for example, welcome their mobile phones being reprogrammed to work like a Motorola phone? These are valid concerns, beyond the scope of this research project to address fully, but they are not unduly worrisome since the task of constraining a powerful system is always so much easier than the opposite task of having to find ways of boosting the power of an inherently weak one.

Reducing the freedom and flexibility that the Mesh provides is just the relatively simple matter of access denial. If the manufacturer of the fan in this example didn’t want changes to be made to how the fan works, they could simply lock the user interface permanently. They would still benefit from having programmed in Meshoil, because whenever changes do need to be made in the future, such as when implementing product revisions and upgrades, making those changes will always be much easier to do. Although the user interface may appear permanently SIC to the user in such a situation, it is more accurate to see it as temporarily frozen, since the inherent flexibility of the system is never lost and is always there to be exploited.

3.7.5 Manufacturer still able to call the tune

The fan manufacturer can set whatever level of flexibility they are comfortable with. They can choose to offer the user:

- no flexibility—by permanently locking the user interface so that it can never be changed once the fan is shipped
- limited flexibility—by allowing users to choose from one of a small number of alternative user interfaces provided exclusively by the manufacturer
- moderate flexibility—by allowing users to build their own user interface in kit form by choosing from a range of software plugins provided exclusively by the manufacturer
- high flexibility—by allowing users to build their own user interface using whatever suitable Meshoil software they find on the Mesh, whether authorised by the manufacturer or not
- unfettered flexibility—by allowing users complete freedom to do anything, including programming the fan by writing their own Meshoil.

Even a manufacturer intent on keeping total control over their products might welcome the second option of limited flexibility listed above, since it would add a great selling point to their product line without any risk to their hegemony.

No method of access denial is proposed in this thesis, but it is not hard to imagine how such a system could work: the Mesh engine loaded in the device produced by the manufacturer might be assigned a hidden ID code. Meshoil could only be downloaded to this device if it was recognised as being officially authorised by containing the same, hidden ID code.
CHAPTER 3. PRINCIPLES OF THE MESH

3.8 “I want thermostat control”

Control settings for this simulation run:

<table>
<thead>
<tr>
<th>Fan1 USER INTERFACE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Connect 3-speed to 5-speed</td>
<td></td>
</tr>
<tr>
<td>2. Black key on/off</td>
<td>Downloaded</td>
</tr>
<tr>
<td>3. White key on/off</td>
<td>Downloaded</td>
</tr>
<tr>
<td>4. Black and white key panning</td>
<td>Downloaded</td>
</tr>
<tr>
<td>5. Thermostat</td>
<td>Downloaded</td>
</tr>
<tr>
<td>6. Panning Irregular</td>
<td></td>
</tr>
<tr>
<td>7. Safety cutout</td>
<td></td>
</tr>
<tr>
<td>8. Noma control from other fan</td>
<td></td>
</tr>
<tr>
<td>9. Clear other fan operation</td>
<td></td>
</tr>
<tr>
<td>10. Only usable if other fan on</td>
<td></td>
</tr>
<tr>
<td>11. Start panning if other fan stops</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.19: Simulation settings: Thermostat plugin

Changes in the Weather Bureau’s temperature readings causing the fan to switch on and off can be simulated by manually changing the setting of the mesh://wb.temperature control towards the bottom of the main control window shown in Figure 3.1 on page 35.

3.8.1 Thermostat plugin

Continuing to browse the ui4u Meshpoint for interesting features to add to his user interface design, the user discovers a useful plugin that adds thermostat control to the fan.

This plugin switches the fan on whenever the temperature rises past a thermostat setting, and switches it off again whenever the temperature drops below the setting. While the fan is off, the whitekey can be used to scroll the thermostat setting in the range 20–50 C, or to switch thermostat control off completely.

He downloads this plugin to his fan. Initially, thermostat control is switched off, so he sets the thermostat to 30. He is pleased to find that on hot, sunny days the fan switches on automatically.

3.8.2 Virtual hardware

How can the fan support thermostat control when it doesn’t have any physical means of detecting the temperature? The list of the five hardware controls—display, motor, pan, blackkey and whitekey—does not include a control such as temperature that we might expect if the device had an in-built temperature sensor.

The fan gets over this problem by getting the necessary information about the ambient temperature from another Meshpoint. It incorporates virtual hardware into the fan by making use of the hardware provided at this remote location.

The wb Meshpoint of the Weather Bureau provides server-side information on the weather as a resource for client-side Meshpoints. The Thermostat plugin makes use of the Weather Bureau’s wb.temperature control that holds the ambient temperature, as measured by a thermometer at the Weather Bureau’s weather station.
Figure 3.20 shows the Thermostat plugin.

Figure 3.20: Thermostat plugin (1 of 3)
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.20: Thermostat plugin (2 of 3)
The Thermostat plugin introduces two new controls to the user interface code: thermostat and wb.temperature. Unlike the five hardware controls, these controls are not defined anywhere in the program so they are created as local programming variables.

Whenever the temperature at the Weather Bureau, referenced by the wb.temperature control, exceeds the setting in thermostat, the fan switches on and starts panning. Whenever the temperature drops below the setting in thermostat, the fan switches off.

3.8.3 Not another power cut!?

Virtual hardware is also the solution to the problem mentioned in section 1.2.2 on page 4 for how to automate the job of resetting the internal clocks of all domestic appliances following a power cut. As Meshable devices, all such appliances would rely on a server-side Meshpoint, similar to the Weather Bureau’s, but one that provided the current time instead of temperature. Each client-side device could choose to have no internal clock of its own and rely totally on the time provided by the service, or perhaps more practically, have an internal clock and just call on the server-side Meshpoint periodically to maintain accurate timekeeping.

3.8.4 Ubiquitous computing

This example of thermostat control shows how the Mesh is able to support the concept of ubiquitous computing, where computers are increasingly interwoven into the fabric of our lives. It is also sometimes referred to as pervasive computing, but this term more accurately describes the use of handheld devices for universal access to services.

The idea is one of embedding very small, networked processors into everyday objects to give them computing power and the potential to interoperate. Ubiquitous computing is becoming increasingly feasible with the continuing advances in microelectronics and network technologies, but still faces the formidable challenge of finding a cohesive, systematic and universal method for allowing devices that may significantly differ in design and function to interact and control one another.

This example demonstrates how the Mesh is able to address this problem by making it easy for devices of any type and description to interoperate via the Internet. Here, the thermometer that
controls fan operation is located at a weather station, but it could equally be well be a miniaturised biometric sensor embedded into the user’s clothing: the principle of disparate types of physical device seamlessly interacting with one another is no different.

3.8.5 Software-based systems

The examples discussed in this chapter are necessarily hardware focused because the device used to demonstrate the various principles of the Mesh is a domestic fan. Such a physical device was chosen because it serves to demonstrate how the Mesh is able to handle the additional complexities of control sharing when hardware is involved. But ‘device’ is used collectively in this thesis to mean software as well as hardware-based systems, and the reader should not be misled into thinking that the Mesh is primarily about the control and networking of hardware.

In the example here, the \texttt{wb.temperature} control accessed over the Mesh happens to be a hardware control. But in many situations, such controls would be abstract in nature without any implied hardware connection.

Consider how a TV station might make available on its Meshpoint a software control called \texttt{NowShowing} that holds the name of the TV program currently being broadcast. Viewers could make use of this information to program their personal organisers to tell them when the programs they want to watch are on. A personal organiser could be set up to display a message whenever the setting for the \texttt{NowShowing} control changed to the name of a desired TV program, and to cancel the message whenever the setting changed from this name to something else indicating the end of the TV program.

3.8.6 Responsibility optimisation

The flexibility of the Mesh means that new efficiencies can be achieved through a process of \textit{responsibility optimisation}: the responsibilities shared by users, programmers and manufacturers are redistributed in such a way to minimise the overall workload and achieve a better solution for all.

Universal usability, discussed in section 3.3.2 on page 41, is an example of this principle. There are benefits to be had by shifting the responsibility of customising a user interface from the manufacturer to the user, representing a shift in the server-to-client direction. Rather than the manufacturer having to face the impossibly huge task of trying to anticipate the needs of every potential user and writing a definitive user interface program to accommodate them all, the responsibility of customising the user interface is best handled by the users themselves, since that is the only way to ensure that they will get exactly what they want. Use of conventional programming tools precludes such a solution because such tools are incapable of producing software that is easily modifiable.

The problem of resetting clocks after a power cut discussed in this section is another example of responsibility optimisation and how the Mesh supports solutions that are impossible to achieve using conventional means. Here, the shift in responsibility is in the opposite direction of client-to-server. Why subject users to all the cumulative time and effort in resetting clocks after a power cut when the task can be handled far more efficiently by having it performed just once at a centralised location to the benefit of everyone?

The TV station example above presents another example of how responsibility optimisation can simplify the user’s life. When the user wants to record a TV program, the conventional solution requires that the user sets up the start and end times of the broadcast in the recording device, either by inputting them manually or by obtaining them in a special TV channel acting as an online TV guide. In the Mesh, this is no longer necessary since timing issues are handled implicitly by
the *NowShowing* control. Getting the user’s recording device to record a TV program by making use of this control is no different from how the user’s personal organiser was set up to indicate when the program is on.

Like the problem of setting the clocks, the efforts of many are replaced by the effort of one. But increased efficiency is only part of what responsibility optimisation can deliver. In this example reliability is increased too, since the responsibility of all issues relating to broadcast timing remain rightfully where they belong—with the TV station. There is never the chance of failing to record a program because scheduling problems at the station result in the program being broadcast at a time different to that set up in the recording device—a common enough occurrence in present-day systems—because at all times there is only one centralised source of such information. And if the TV station has chosen to indicate commercial breaks as well as programs in *NowShowing*, perhaps as a chargeable service to users, uninterrupted program viewing will be a further benefit since recording will automatically stop during these breaks.
CHAPTER 3. PRINCIPLES OF THE MESH

3.9 “I want a sleep mode that times out panning”

Control settings for this simulation run:

<table>
<thead>
<tr>
<th>Control</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
<tr>
<td>Downloaded</td>
<td>Downloaded</td>
</tr>
</tbody>
</table>

Figure 3.21: Simulation settings: Panning timeout plugin

3.9.1 Panning timeout plugin

The next plugin our user downloads and adds to his user interface is a sleep mode feature that automatically switches panning off after panning has been running for a specified time.

Figure 3.22 on page 71 shows the Panning timeout plugin. Like the previous example of the Thermostat plugin, the user is able to switch the feature on and off, and select different settings. While the fan is off, the whitekey can be used as a shift key with the blackkey to set the timeout to a value in the range 10–60 seconds, or to switch the timeout feature off.

3.9.2 Method calling

How is the fan able to support timed operations? Many electronically controlled devices might support some sort of timer function in their firmware based on their internal clocks, but this is not the case here. The device is just a simple, electronically controlled fan.

The Panning timeout plugin introduces two new controls to the user interface code: timeout and timer. The timeout control holds the user specified setting while the timer control provides the actual timing. When panning starts, the plugin sets the timer to start in order to start this control counting up in seconds. When the value of timer reaches the value specified in the timeout control, for example 20 for 20 seconds, the plugin stops the fan panning and resets the running of the timer by setting it to stop.

The plugin does not know—nor needs to know—how the timer control works, since this is handled elsewhere. The plugin simply sets timer to the character string ‘start’ and waits until it changes to the required numeric value. The timer control is one example of how a Meshoil program makes the equivalent of a method or function call, triggering the external execution of code. In a conventional language, timer would be a method supported by the language’s API (Application Programming Interface). The character string ‘start’ would be passed across in a call to this method and the values returned by the method monitored until the correct one was reached.

But in the example here, the platform is just the very simple one of a fan device. It doesn’t support an API that one would expect of a full-featured, programming platform. So how is API support still possible for the Panning timeout plugin?
3.9. “I WANT A SLEEP MODE THAT TIMES OUT PANNING”

Figure 3.22: Panning timeout plugin (1 of 3)
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.22: Panning timeout plugin (2 of 3)
3.9.3 Ghost-in-the-machine API

In Meshoil, methods are supported three ways:

- from the *ghost-in-the-machine API* Meshpoint that acts as universal, online API resource for any Meshoil program connected to the Mesh

- from the API provided by the platform where the Mesh engine runs, if such an API is available (for example, the Java API for Mesh engines written as Java applications—as is the case for the concept demonstrator)

- as local, user-created methods, written in Meshoil and included as part of the program making the method calls.

The first alternative of the ghost-in-the-machine API is the one used here (—or at least simulated! Code for supporting the *timer* method is included in the 2Fan.xml file). The full name of the *timer* control used in the plugin is `api.timer`, representing the *timer* method provided at the `api` Meshpoint—the ghost-in-the-machine API.

Support for the ghost-in-the-machine API in the Mesh means that API resources such as method libraries do not have to be physically located where they are required by an executing Meshoil program. No local support of API functionality is necessary since, so long as the Mesh engine has Internet access, all API resources are available at a single, centralised, universal location that serves the whole of the Mesh.

Use of the ghost-in-the-machine API has the effect of instantaneously upgrading an electronic device, no matter how basic, to the level of a programming platform.

In practice, the Mesh might support a number of such ghost-in-the-machine API Meshpoints. To simplify programming in Meshoil, they might all have the same Meshpointer of `api`, with
CHAPTER 3. PRINCIPLES OF THE MESH

transparent routing mechanisms being used to determine how these special, server-side Meshpoints share the client load and collectively respond to all API method calls made across the Mesh.

3.9.4 Abstract API layer

Programs written in Meshoil retain their platform-independence regardless of what platform the Mesh engine runs on, even when they make use of the API of the Mesh engine’s platform. Universal code can be written that never needs to be edited to suit the particular language that was used to create the Mesh engine application.

Take the example of the concept demonstrator version of the Mesh engine, which is a standard Java application running on the Java platform. If a Meshoil program run by this Mesh engine chose to make use of the platform’s API rather than relying on the ghost-in-the-machine API, the API would be the standard Java language API. The program would be able to make full use of the Java API without compromising its platform-independence. Should it need to be run by a Mesh engine written in another language—a Mesh engine written as a C++ application for example—it would work just as well as before, unaware that its method calls were no longer being serviced by the Java API but by the API of another language.

What makes this possible is Meshoil’s abstract API layer that provides a flexible way of mapping abstract methods used in the Meshoil program with specific methods provided by the platform’s API. In the same way that the abstract hardware layer allows a Meshoil program to control hardware without any knowledge of how the hardware physically works, the abstract API layer allows a program to access a platform’s API without knowing anything about the names of API methods or the arguments they take. It is not even necessary for there to be a direct, one-to-one correspondence between an abstract method and its API equivalent (for details on the abstract API layer and the various ways that Meshoil supports method calling, see appendix A.5 on page 307).

3.9.5 Engineering integration by layering

Many solutions to complex engineering problems are achieved by breaking down one large problem into a number of smaller, more easily solved problems. Each smaller problem is made self-contained by clearly separating out duties and responsibilities. This makes the task of assembling the individual solutions to solve original problem that much easier. The process often involves a layering approach in which the individual solutions are built one on top of another.

Meshoil follows this model for engineering integration by layering by forming a discrete functional layer. It achieves self-containment by the abstract nature of its syntax and the way that it provides a number of well-defined handles for interconnection with the bigger engineering picture.

When Meshoil is used for a user interface application such as this one, the user interface program forms a layer sandwiched between the abstract hardware and API layers. The abstract hardware layer above the program provides the connection to the device that the user sees and operates, while the abstract API layer below the program provides the low-level functionality for supporting the running of the program.

It is this structural arrangement that underpins the use of Meshoil as a language for universal usability.
3.10  “I want the fan to operate safely”

Control settings for this simulation run:

![Simulation settings: Safety cutout plugin](image)

3.10.1  Safety cutout plugin

The final plugin our user downloads from the ui4u Meshpoint and adds to his user interface is a safety cutout feature. This feature automatically switches the fan off if any electrical fault is detected in the fan. Figure 3.24 on page 76 shows this plugin.

3.10.2  Team effort in program design

The Safety cutout plugin uses information about fault conditions that are provided at another Meshpoint, the wilsonelectric Meshpoint of the fan manufacturer.

The shareware programmer, who wrote the Safety cutout plugin and uploaded it to the ui4u Meshpoint for the benefit of all other shareware users, knew what she wanted the fan to do if faults were detected—namely, immediately warn the user and switch everything off—but lacked sufficient knowledge about the device to know what all the fault conditions might be. Only the manufacturer of the fan would know this type of specialist information. The shareware programmer decided to write the plugin by making the cutout procedure dependent on the list of fault conditions identified by the manufacturer, whatever they might be.

At the manufacturer’s Meshpoint, two fault conditions for the fan are described. Firstly, the panning solenoid should never be energised while the fan isn’t running, since this wastes electricity pointlessly. And secondly, for the purpose of detecting current leakage problems, the motor should never be on at a setting so low that it is unable to make the motor shaft physically turn even though the motor is drawing a current.

3.10.3  Brick nesting provides software scalability

Software scalability is the ability of a programming language to handle increasingly large programs in a graceful manner. In the Safety cutout plugin, scalability is demonstrated by the way that the layering concept, used in the creation of Meshoil, is applied to the code within a program. The Meshoil program, forming a layer in the bigger engineering picture, is itself internally layered to represent the responsibilities of the various tasks that the program must perform.
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.24: Safety cutout plugin
By following this approach, a Meshoil program is able to support any level of design granularity required for the programming solution. The mechanism used to connect the layers and create the resultant, functioning program is brick insertion. The program is built up in nested stages by assembling larger and larger blocks of code that are responsible for progressively higher and higher programming tasks. At each level, bricks have server relationships with their parental bricks above, and client relationships with their child bricks below.

In the example here of the safety cutout feature, program layering consists of three levels:

- **Top level (user)**—the main body of the fan user interface code
- **Middle level (shareware programmer)**—the **Safety cutout plugin** inserted into the main body of the fan user interface
- **Bottom level (manufacturer)**—fault condition code inserted into the **Safety cutout plugin**.

This nested layering is visible in the **Safety cutout plugin** on page 76. Bottom level bricks called `mesh://wilsonelectric/fan.faultcondition` are inserted into the `mesh://ui4u/fan.plugin.safetycutout` middle level brick, which in turn is inserted into the top level code of the user interface.

### 3.10.4 Software-team scalability

The **Safety cutout plugin** shows how software scalability isn’t just a feature of the program, it is reflected in those writing the program too. Traditionally, writing a program is the task of a highly coordinated and planned team with sophisticated synchronisation tools, but the Mesh principle of **software-team scalability** means that everyone can join in and play their part according to their own particular interests and abilities. The bigger the program, the greater the number of people that may be involved and the greater the helpful division of labour (another example of responsibility optimisation that was discussed in section 3.8.6 on page 68).

Unlike a conventional programming team, there isn’t a need for the team players to be in close connection with one another, whether geographically or through sharing a common goal. Nothing is required other than a clear understanding of individual responsibility. This is shown in this example by the way that layering within the program is a direct reflection of the knowledge and responsibilities of the people involved in its creation.

The fan user at the top level is not interested in anything other than knowing that adding the **safety cutout plugin** to the user interface will make fan operation safer. His task is simply to download this plugin from the ui4u Meshpoint and insert it in the user interface.

The task of the shareware programmer at the middle level is to flesh out the one-line mission statement for safer fan operation that caught the eye of the fan user. She creates a plugin that will output a message on the fan’s display and switch everything off if any faults are detected. She does not need to know how many fault conditions there are or what each one entails, since this level of detail is left to somebody else to handle. All she needs to do is to include a single reference in the plugin for the insertion of any bricks called `wilsonelectric/fan.faultcondition`.

At the bottom level, the fan manufacturer meets its responsibilities to all those involved in use of its fans—users and shareware programmers alike—by writing code that describes all known fault conditions for the fan hardware and making this code available on the company’s `wilsonelectric` Meshpoint. The manufacturer doesn’t know how this code will be used or by whom, but this is of no importance.
3.10.5 Cut and paste programming

Meshoil’s minimal-rule syntax makes it a robust language for manipulating code and for cut and paste programming. There are fewer rules governing how programming constructs have to be defined and positioned within the code.

Programming in a conventional language is less forgiving, since the rules governing syntax and associations between code constructs are more complex. Successful programming demands greater skill and knowledge of the particular idiosyncrasies of the language, if compilation and runtime errors are to be avoided.

As the Safety cutout plugin shows, code can be freely pasted almost anywhere within a Meshoil program. Code is effectively pasted from one Meshpoint location right into the middle of an IF statement condition defined within a program at another Meshpoint location. The plugin writes the first half of the IF statement condition as ‘if any...’. Instead of completing the condition by explicitly writing what the conditions are, it makes reference to a brick called \texttt{wilsonelectric/fan.faultcondition}, effectively relying on the manufacturer to write the second half of the IF statement condition.

Figure 3.25 on page 79 shows the fault condition information provided by the \texttt{wilsonelectric} Meshpoint. Two bricks have the name \texttt{wilsonelectric/fan.faultcondition}. The first of these uses two nested bricks to define the panning solenoid fault as the \texttt{pan} control being \texttt{On} while the \texttt{motor} control is \texttt{0} (motor off). Use of the all spin control indicates that for the fault condition to be true, both of the nested bricks must be true—ie, \texttt{pan} must be \texttt{On} and \texttt{motor} must be \texttt{0}. The second brick also uses two nested bricks. It defines current leakage as the \texttt{motor} setting being greater than \texttt{0} but less than or equal to \texttt{0.001} (motor on at any setting up to 0.1% of its maximum).

In a conventional language, the combined IF statement would be expressed using a combination of nested brackets and AND/OR logical operators in the following format:

\[
\text{IF } (((\text{pan} == 'On') \text{ AND } (\text{motor} == 0)) \text{ OR } ((\text{motor} > 0) \text{ AND } (\text{motor} <= 0.001)))
\{
\text{display} = 'Fan fault! Safety cutout!';
\text{motor} = 0;
\text{pan} = 'Off';
\}
\]

Even if the conventional language supported a generalised method for inserting code such as Meshoil’s brick insertion mechanism, it might be fiddly to use in cases like these because of the muddled mix of hierarchic (ie, brackets) and sequential (ie, logical operator) syntactic styles. Individual fault conditions might need to be carefully inserted between repeating instances of OR logical operators.

The programmer’s task in Meshoil is an easier one. All the necessary fault conditions are able to be pasted in at a single location underneath the stipulated ‘if any...’.

Two things make this possible. Firstly, the strictly adhered to, consistent nature of Meshoil’s hierarchic syntax means that it is always possible to maintain a clean break between server-side and client-side code. And secondly, Meshoil replaces the use of the conventional AND/OR logical operators with more powerful and generic equivalents—the all and any spin controls (see appendix B.2.1 on page 325 for a discussion on this).

Some programming experts believe that programming by pasting code is bad practice, since the very fact that it is so quick and easy to do increases the chance of insufficient care and attention being given to the task of programming. But this concern usually refers to the manual process of pasting. While Meshoil supports manual pasting too, the example here is about automated pasting. The program has been set up so that the actual pasting of code happens automatically.
when the program runs—the program effectively stitching itself together in a carefully, preordained manner.

**Figure 3.25: Wilson Electric Meshpoint**
3.10.6 Globally-unified hardware control

While electrical devices are built to serve many different functions, they often share similar hardware features. For example, many devices will have a DC power supply circuit able to be represented in a Meshoil program by a control called \texttt{Power} that takes \texttt{On} and \texttt{Off} settings. These devices may feature a number of push-button keys that can all be represented by controls with identical hardware descriptions. In devices like mobile phones, even the number of these keys and their names will be known.

The Wilson Electric company produces a range of household appliances—such as fans, vacuum cleaners, microwave ovens and washing machines—all of which contain electric motors. These motors may all differ in terms of:

- Physical size (small, medium, large)
- Input supply type (DC, AC)
- Motor type (DC, synchronous, induction)
- Rotor type (squirrel cage, wound rotor, salient pole, PM)
- Rated quantities (speed, torque, current, voltage).

Yet, despite the wide range in such physical characteristics, all these motors remain identical at the abstract level of being a ‘motor’. In Meshoil they can all be represented by a control called \texttt{motor} that takes a setting anywhere in the normalised range of 0 (standstill, motor off) and 1 (rated speed, motor on maximum)\textsuperscript{1}.

Meshoil’s abstract hardware layer thus acts as a great leveler, standardising hardware from the real world by simplifying how it is viewed from within a program. For the purpose of controlling such hardware, the many differences in physical characteristics can be safely ignored. This allows the Mesh to focus on functional similarities and exploit the great commonality between devices of different type. Not only will this tend to make it easier to program any single device, since there is a common feel to the programming task, but we might expect interoperation between devices of dissimilar nature to be easier to achieve too.

Wilson Electric takes advantage of this Mesh principle of \textit{globally-unified hardware control}. At their Meshpoint, they provide a single definition of current leakage can be used for any of their motorised products.

In support of globally-unified hardware control, a global registry would be set up for managing commonly used control names in the Mesh and their definitions, in much the same way that URL addresses are assigned and registered for the Web.

3.10.7 Maximising code reuse

Various brick naming schemes can be used to maximise code reuse in Meshoil programs. Here, the \texttt{wilsonelectric} Meshpoint increases code reuse by a naming scheme that not only gives bricks more than one name but also gives the same names to more than one brick. This benefits the manufacturer and shareware programmer alike.

For example, had the shareware plugin in this example been for a vacuum cleaner not a fan, a single reference for the insertion of any bricks called \texttt{wilsonelectric/vacuum cleaner.faultcondition} would have resulted in inserting two conditions: the same current leakage fault used for the fan,

\textsuperscript{1}Meshoil’s abstraction of motor speed in the normalised range of 0–1 is, in fact, the reciprocal of the term ‘slip’, used to describe how close to synchronous speed an induction motor runs at.
but the dust bag fault shown in Figure 3.25 instead of the panning solenoid one. The manufacturer would not have needed to double up on the definition of current leakage for their vacuum cleaners, and the shareware programmer would still have managed to insert everything needed to describe vacuum cleaner fault conditions with a single reference.

If the shareware programmer wanted to know about fault conditions applying to any type of motorised device manufactured by Wilson Electric, she could have referred to a brick with the name \texttt{wilsonelectric/motoriseddevice.faultcondition} that would have successfully picked up the generic current leakage fault, but none of the unwanted, device-specific ones.

By carefully choosing how bricks are named, the \texttt{wilsonelectric} Meshpoint is able to standardise its server-side library of fault conditions and avoid code duplication without any loss of personalisation offered client-side to users of their products.
3.11 “I want the latest plug-and-play design done for me”

Control settings for this simulation run:

```
- fan user interface
  - Current speed is 5-speed
  - 1. Black key switch
  - 2. White key switch
  - 2. Black and white key pressing
  - 4. Thermostat
  - 5. Panning banned
  - 6. Safety cutout
  - Remote control from other fan
  - Clone of other fan operation
  - Only usable if other fan is on
  - Start fan if other fan stops panning

Figure 3.26: Simulation settings: Latest design
```

3.11.1 Latest design

Our user is happy with all the plugins he has downloaded, but one nagging thought remains. What if someone later uploads a plugin for a great new feature that he might really like to include in his user interface design, if only he knew about it? While he can keep up to date with the latest features available at the ui4u Meshpoint by checking it out every once in a while, what he would really like to do is to automate the process. By doing this, he could be sure that he was always using the latest and most comprehensive user interface design.

Then he realises that this is made possible for him because all the really useful plugins at the ui4u Meshpoint have been given the common name of ui4u/fan/latestdesign in addition to their own individual names. Rather than doing all the hard work himself by downloading individual plugins, he sets up the user interface as a single brick that references this common name. From then on, every time he switches the fan on and execution of the user interface code begins, the complete set of plugins automatically downloads to create a composite user interface containing all the latest features.

Better still, he gives this solitary brick online spin, so that even if new plugins are uploaded to the ui4u Meshpoint as his fan is running, he will immediately benefit from them. His user interface will now update itself, even as the fan is running.

He knows that if changes are made to his user interface that he needs to know about, such as the inclusion of a new user feature, he will immediately be advised of what the changes mean to him. For example, an email would accompany the sudden inclusion of the Thermostat plugin describing how to make use of this new feature. But changes that won’t have any impact on how he controls the fan, such as the inclusion of the Safety cutout plugin, will go unannounced because he has already told ui4u that he only wants to be notified of changes that affect him directly.

3.11.2 Freedom to restructure a Meshoil program remotely

The Thermostat plugin showed how a Meshpoint can reference information about controls found at another Meshpoint, the particular example being a control that represented the temperature. The Safety cutout plugin then showed how the method is exactly the same when referencing information about bricks instead of controls, the only difference being that the information downloaded from the remote location to the program forms a complete block of code rather than the setting of a single programming variable.
In a conventional language, it is easy to write a program that reads in external values that modify how the program runs, but far from easy to write the program in such a way that the external information can directly modify the code of the program. In Meshoil, it just as easy to do either.

Was the Safety cutout plugin an example of remote information changing how the plugin ran, or changing how it was written? Since the brick was inserted from the manufacturer’s Meshpoint to form part of a conditional expression, it could be argued that it didn’t actually change what the plugin does, only when it does it. But had the brick been inserted outside of the conditional expression as a separate piece of executable code in its own right, it would clearly have changed the very fabric of the plugin.

In the example here of the latest user interface design, remote information doesn’t just change how the program is structured, it defines the program completely, since the program consists of nothing else but inserted code.

3.11.3 Timing of information transfer across the Mesh

When the information accessed from a remote location is a control setting, the setting is downloaded during runtime. A download operation is triggered each time the part of the program containing the reference is executed. This ensures that the control setting used in the program is kept up to date with the actual control setting at the remote location. The control remains effectively online like this for as long as the program continues to execute.

For bricks, however, the default is for brick information to be downloaded offline, since it is not always necessary—or even desirable—to have program code constantly being subject to modification during execution.

A brick can thus be downloaded to a program from another Meshpoint:

- manually on command
- automatically at compilation time or when the program starts to run (default, offline spin)
- automatically during program runtime (online spin).

3.11.4 Self-updating programs

The Safety cutout plugin didn’t specify any spin when referring to the bricks at the wilsonelcetrc Meshpoint, so the offline default was used. Every time the fan is switched on, the latest information about fault conditions from this Meshpoint will automatically download to the fan and be used for the duration that the fan remains running. The fan’s safety cutout feature is thus self-updating. Should the manufacturer revise the list of fault conditions at any time, the changes will automatically become incorporated in how the fan works the very next time it is switched on.

In the example here, however, the fan user specifies online spin for the single client-side brick containing the reference to mesh://ui4u/fan.latestdesign, so that even if changes happen to the information at the remote location after the program has started executing, they will immediately be incorporated in the running program. When online spin is specified, the reference to the code to be inserted acts more like a dynamic pointer instead of a copy and paste command, since the information downloaded during program execution is transient and constantly being refreshed.

In practice, it would rarely be necessary to download information in every program execution cycle since it would be unlikely to be changing so quickly. The mechanism to support online spin might involve caching. Code would be repeatedly downloaded from the remote location and cached, but not necessarily at such a high frequency as that of the program execution cycle. Whenever a
cached version was found to have changed from that of the previous download cycle, the cached version would be physically inserted into the executing program, overwriting the existing version.

This approach would reduce the occurrence of repeatedly transferring information that has not changed. If found to be necessary, the problem could be avoided altogether by implementing transparent mechanisms for supporting this programmer-level concept of polling for remote information: the client-side could request the server-side only to transfer the requested information at the point that it is found to have changed client-side.

3.11.5 Program morphing

The self-updating program for the latest plug-and-play design of fan operation is an example of program morphing—the process by which the executable version of a program is rewritten on-the-fly in realtime as the program is executing.

If a program contains a number of references to bricks at remote locations, these references are given online spin, and the bricks are subject to modification at the remote locations, then the running program can systematically and purposefully undergo change, morphing into a very different program without any apparent intervention. But program morphing does not always have to be as dramatic like this. For example, it could be the mechanism used for something as simple as creating a phone book entry in a mobile phone user interface. Rather than assigning data to memory already set up for the purpose, the user interface code would physically grow through the addition of bricks to accommodate the new information.

It is easy for the Mesh engine running the program to support program morphing because the process involves the standard OO programming techniques of runtime class instantiation and garbage collection. A Meshoil program is just a collection of brick objects, and the Mesh engine is little more than a software application for creating a single type of object by instantiating the brick class (see appendix C.1 on page 335). In a conventional language, nothing usually happens once a class has been instantiated until methods are called. In Meshoil, however, program execution immediately changes on brick instantiation because the brick object is a piece of executable code.

3.11.6 Self-generating software

The example mentioned above of creating a phone book entry introduces the idea of self-generating software in the Mesh, an interesting concept but one that is beyond the scope of this research project to explore.

Self-generating software is an area of computer science often shunned because of its complexity. When a program has the ability to modify itself, the resultant behaviour of the program can be hard to predict. But as the phone book example shows, the technique could be put to good use for quite mundane, practical and predictable purposes. All that would be necessary is to expand Meshoil’s existing brick insertion mechanism to allow bricks to be inserted in other bricks at runtime. Meshoil syntax would need to be able to specify the exact nature of the bricks to be inserted, where they are to be inserted, and under what conditions they would be inserted into the existing code of an executing program.
3.12. “What use is a fan in winter?”

Control settings for this simulation run:

```
<table>
<thead>
<tr>
<th>USER INTERFACE</th>
<th>Stopwatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>--Connect 3-speed to 5-speed</td>
<td>-</td>
</tr>
<tr>
<td>--1. Black key on/off</td>
<td>-</td>
</tr>
<tr>
<td>--2. White key 5-speed</td>
<td>-</td>
</tr>
<tr>
<td>--3. Black and white key panning</td>
<td>-</td>
</tr>
<tr>
<td>--4. Thermostat</td>
<td>-</td>
</tr>
<tr>
<td>--5. Panning timeout</td>
<td>-</td>
</tr>
<tr>
<td>--6. Safety (old)</td>
<td>-</td>
</tr>
<tr>
<td>--Remote control from other fan</td>
<td>-</td>
</tr>
<tr>
<td>--Change of other fan operation</td>
<td>-</td>
</tr>
<tr>
<td>--Only usable if other fan is on</td>
<td>-</td>
</tr>
<tr>
<td>--Start panning if other fan stops panning</td>
<td>-</td>
</tr>
</tbody>
</table>
```

Figure 3.27: Simulation settings: Stopwatch user interface

3.12.1 Stopwatch user interface

Winter approaches and our user has no further need of his fan until spring next year will herald the return of warmer weather. What to do with the fan in the meantime? He finds it cluttering up the kitchen for no useful purpose. He thinks all this hardware must be good for something, and then gets the brainwave of using it for doing something quite different from just keeping the kitchen cool.

Why not use it as a stopwatch for timing how long to simmer his winter casseroles? After all, the fan unit has a display and keys, which is all that he really needs, and he knows that it is able to handle timed operations from the way that the Panning timeout plugin worked. The moment the hot weather returns, he can switch it back to being a fan again.

His idea is so original that he can’t find a suitable user interface on the ui4u Meshpoint to do the job. Undeterred, he goes to the Google Gizmo Meshpoint to see if he can find one by doing a search.

Google Gizmo is a special Meshpoint that closely follows the model of Google’s Web search engine. You enter criteria about the device to be controlled, such as its make and model number, together with the user user features you want. The results of the search list all programs found on the Mesh that most closely match what you are looking for. Should the device be currently connected to the Mesh, the search process is even simpler. Instead of having to describe the device, you only need to enter its Meshpointer. This allows the search engine to identify online details about the device and how it is currently being used, allowing the search to be refined even further to match your likely needs. All you have to do in order to install new software for the device is to click on any of the links returned by the search. This runs a wizard to download and install the software automatically.

But sadly, even after an exhaustive search on the Mesh, our user still can’t find the user interface he is after. This leaves him one remaining option—write the user interface himself! Emboldened by his experiences through the summer of downloading user interfaces and plugins, he thinks that writing the user interface himself can’t be so hard. He uses one of the standard Meshoil builder GUI tools to write the Meshoil code, a job known in the vernacular as ‘producing oil’.

---

2Google [15] is the company world-renown for its search engine but Google Gizmo is, of course, hypothetical.

3No such tools have been implemented in the concept demonstrator, but given the simple, modular structure of Meshoil these tools might be relatively easy to develop. They would save the Meshoil programmer from having to hand code at the XML editor level, perhaps similar in design to the way that Web page creation tools save the HTML programmer from having to write code at the text editor level.
Pleased with how his design works, he names it the **Stopwatch user interface** and uploads it to the **ui4u Meshpoint** so that other fan users can benefit from what he has achieved.

Figure 3.28 on page 87 shows the **Stopwatch user interface**.

### 3.12.2 Hardware reuse

Meshoil supports the concept of software reuse like any other OO programming language (see appendix B.3.3 on page 330), but it goes beyond this to support the concept of **hardware reuse** as well.

Hardware reuse is another benefit of the expanded solution space provided by Meshoil’s hardware abstraction layer. Existing hardware features of a device are able to be reused so freely, that they provide the user with what appears to be quite a different type of device. In the example here of such a simple device as a fan, there is little scope for imaginative hardware reuse. But at least the display and keys can be put to good use in serving other purposes.

When used as as a stopwatch in this example:

- the **blackkey** starts and stops the time counting in seconds
- the **whitekey** resets the time to zero
- pressing the **whitekey** a second time switches the stopwatch off.

Like the **Panning timeout plugin**, the **Stopwatch user interface** makes use of the **timer** method supported by the ghost-in-the-machine API at the **api Meshpoint**. The **motor** and **pan** controls of the fan are simply ignored.

For a more interesting example of hardware reuse, see section 5.7 on page 217. It shows how a mobile phone handset operating as a Nokia 6210 phone can be converted to work as an emergency pager for a disabled user.
Figure 3.28: Stopwatch user interface (1 of 2)
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.28: Stopwatch user interface (2 of 2)
3.13 “I want to run all these user interfaces”

Control settings for this simulation run:

<table>
<thead>
<tr>
<th>User Interface</th>
<th>All</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard (1-speed, pan)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Simplified (1-speed)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Standard (1-speed, pan, LED)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Advanced (3-speed, pan)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Advanced (5-speed, pan)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Latest design</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stopwatch</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Finding it hard to choose which user interface he really wants from all that he has tried, our user finds the solution to the problem at the u4u Meshpoint: all available user interfaces conveniently bundled into one package that allows any one of them to be selected each time the fan is switched on. He is pleased to see that his Stopwatch user interface is included.

The software bundle comprises the following user interfaces:

- Standard (1-speed, pan)
- Simplified (1-speed)
- Standard (1-speed, pan, LED)
- Advanced (3-speed, pan)
- Advanced (5-speed, pan)
- Latest design
- Stopwatch.

When the fan is off, the first press of the blackkey displays the name of the currently selected user interface. The whitekey can then be used to scroll the user interfaces, each key press bringing up the next one in the list. Pressing the blackkey again completes the selection procedure and allows the keys to be used in the normal way for operating whichever user interface has been selected.

The All user interfaces bundle consists of all the code from the individual user interfaces, plus additional code for handling the user interface selection procedure.

3.13.2 Interface agents

What is the best way to operate a user interface? Allow the user to maintain full control at all times or allow interface agents to make autonomous decisions designed to anticipate the user’s needs and make life easier? This question is the focus of much ongoing debate because interface agents tend to violate good usability principles such as predictability and scrutability.
An interface agent aims to boost the prominence of those aspects of a user interface that it thinks will be of interest to the user. Information collected about user behaviour is read into a model of the user to decide on the autonomous actions to take. In the world of knowledge-IT exemplified by the Web, this may involve nothing more than the presentation of information, for example, proactively displaying links for products and services likely to appeal to the user. But in control-IT, the task is likely to be far harder because it involves changing how a device operates, whether software or hardware related.

Take the example here of the device being the physical one of a fan. If an interface agent detected that the number of user features was too many for the user to cope with, a way of simplifying the user interface by reducing this number would be needed. If the user were to be observed regularly switching the fan on at a certain ambient temperature, means would need to be provided in the user interface for automating this behaviour.

### 3.13.3 Problems faced by conventional interface agent-based systems

In a conventional interface agent-based system, autonomous adaptability is usually achieved by writing a sophisticated algorithm for modifying user interface operation and incorporating it within the user interface program itself. There are, however, a number of problems with this approach:

- It means considerable programming effort
- It may fail to be a comprehensive solution
- Scrutability is made more difficult
- It only addresses the customisation issue for one type of device.

Considerable programming effort is involved because the algorithm has to anticipate all the ways that customisation might need to occur. Given that the needs and operating patterns of every user differ, a comprehensive solution to the problem is unlikely to be attainable. Indeed, in some cases the task may prove to be impossible, as demonstrated in the example of temperature-related fan operation where no amount of autonomous tweaking of the user interface can overcome the fact that the device itself has no means of detecting the temperature.

Scrutability—a major concern with all agent-based systems—is made more difficult because any attempt in understanding how the user interface is liable to undergo change requires insider knowledge about the design of the particular algorithm used in the software.

And finally, since devices vary in form and function, all these problems are compounded by the fact that the job of designing, writing and implementing the algorithm has to be repeated for each different type of device.

### 3.13.4 Universal agent

The Mesh avoids these problems by using a *universal agent* capable of autonomously customising the operation of any type of device. There is no need to write an algorithm for modifying a user interface to work a certain way since, in all probability, the work has already been done in the form of an existing version of the user interface somewhere in the Mesh that meets the requirement.

The large resource base of the Mesh is thus exploited to save the programmer from constantly having to reinvent the wheel. This is demonstrated in the fan example, where the task of either simplifying the user interface or of automating temperature-related operation is accomplished in the single step of switching to a ready-made solution (either the solution described in section 3.3.1...
3.13. “I WANT TO RUN ALL THESE USER INTERFACES”

The temperature-related example also demonstrates how the universal agent is able to customise a user interface in ways that are impossible for a conventional interface agent, since no solution would be possible here but for the use of virtual hardware.

3.13.5 A wiki approach for agent-based systems

The universal agent can be seen as applying the wiki model to agent-based systems. The universal agent is the ‘Web 2.0’ upgrade of the ‘Web 1.0’ conventional interface agent. The conventional interface agent is faced with the job of having to do all the work itself in adapting how the user interface works. The universal agent makes all this hard work unnecessary by leveraging the past efforts of others.

The universal agent thus replaces the task of writing code with the much simpler one of using a search engine to identify where the ready-made code exists. The various mechanistic ways in which code can be uploaded to the device, such as whether the code represents the complete user interface or just part of it in the form of a plugin, have already been demonstrated in the previous sections in this chapter on the different strategies for customising the fan user interface. The reader has only to imagine the decisions made and actions taken by the user in these examples being replaced by those of a search engine and automated installation procedure.

The creation of a universal agent-based system for the Mesh was even hinted at in the example of the Google Gizmo Meshpoint that provided the user with a list of alternative user interfaces most closely matching the user’s requirements (section 3.12.1 on page 85). The user had the final say in which one to pick, but the process of finding the likely best match was agent controlled, based on knowledge of the device to be controlled and the user having indicated the desired user features. It is only a small step to go to the next stage and fully automate the selection process, removing the need for any direct input by the user. Instead of basing the user model on the user’s entered selection criteria, knowledge obtained from the user’s past preferences and behaviour would be used. Instead of manually running the search engine at a Meshpoint such as Google Gizmo, an application performing the equivalent function would run in the device itself as an autonomous, background application.

3.13.6 Improved scrutability

The universal agent would tend to reduce the scrutability problem for agent-based systems for two main reasons. Firstly, in having been designed and written for a clear purpose, the characteristics of the ready-made code used for the customisation would likely be well-known, described and documented. There would be no need to interpret modifications made to code by an arcane algorithm. Secondly, there would only be a single user model to understand since it would form an integral part of a universal search engine serving the whole of the Mesh. As necessary, tools could be provided for revealing details of the model’s design and the information gathered about each user which the model uses.

The design of software search engines used in the Mesh might closely follow their Web counterparts. Meshoil programmers would include in their code keywords such as ‘expert user’ and ‘haptic feature for vision-impaired’, designed to be picked up by Mesh crawlers for subsequent search engine use. As well as such explicit cues, implicit cues would be used to enhance search engine performance. For example, user interfaces observed being favoured by vision-impaired users
might automatically become identified as ‘vision-impaired suitable’.

3.13.7 Freedom in meeting the user’s needs

The examples that have been discussed so far in this chapter demonstrate how users have complete freedom over the process of customising how a Meshable device is to work, whether by direct manipulation or with the assistance of an interface agent.

We have seen how users can:

- Accept how the device works the moment it is taken out of the box and change nothing
- Choose from a small selection of user interfaces provided either by the device manufacturer or a trusted shareware Meshpoint
- Choose from the full range of user interfaces available on the Mesh by using specialised Meshpoints, similar in function to enhanced Web search engines, that find and install user interfaces according to user preference
- Build the user interface themselves in DIY kit form by choosing a selection of pre-designed modules
- Rely on a trusted Meshpoint to assemble and choose the latest version of a user interface for them each time the device is switched on
- Write the code for the user interface themselves—a process made easy by the use of standard GUI programming tools
- Hedge all bets on what user interface to choose and get the device to offer a selection each time the device is switched on.

Control settings for this simulation run:

![Simulation settings: Remote control](image)

3.14.1 Remote control

Summer returns with a vengeance and our user invests in a second fan. Keeping the original fan1 in the kitchen, he sets up the new fan2 in the bedroom. He likes the idea of linking the two fans so that whenever he operates the bedroom fan, the one in the kitchen will work too by remote control. He finds the Remote control from other fan plugin at the ui4u Meshpoint, downloads and customises this plugin, and installs it in the fan1 user interface.

When he goes to switch on the fan in the bedroom, he finds that the kitchen fan has switched on too.

3.14.2 Program interaction across the Internet

In the Thermostat plugin and Panning timeout plugin examples of client-server relationships between Meshpoints, the fan used the services provided by the Weather Bureau and ghost-in-the-machine api Meshpoints. These two server-side Meshpoints acted as universal resources for any client-side Meshpoint to use. But any type of relationship between Meshpoints is possible, as demonstrated here by the way that two Meshpoints interact with each other on an individual, personalised basis.

Any Meshoil program can be made to interact and interoperate with another, so long as they both have an Internet connection.

3.14.3 Superprogramming

By combining the global connectivity of the Internet with a universal programming language that supports atomic-level networking, the Mesh provides unlimited flexibility for device interoperation
CHAPTER 3. PRINCIPLES OF THE MESH

or superprogramming\(^4\). Any number of devices can be connected in this way with any degree of interaction, from devices running almost autonomously with minimal influence on each other, to devices exhibiting complete interdependence and functional integration.

Pushing the concept to its theoretical limit, the software controlling all Meshable machines, devices and systems in the world could be combined into one, physically distributed, but globally integrated, executing superprogram.

Although this example and the ones that follow discuss the simple case of superprogramming two identical devices, it should be remembered that there isn’t any requirement for devices to be of similar type and function, or for their interacting programs to be in any way similar in design or purpose. Our fan user could equally well have linked operation of his kitchen fan to that of his microwave oven, or indeed to some piece of software running at a location in the Mesh that had nothing to do with the programming of user interfaces at all.

3.14.4 Customising the plugin

None of the plugins discussed so far have required editing after being downloaded. Each one was a self-contained, plug-and-play module. The names of controls contained in the plugin were automatically customised for operating the user’s fan by simply placing the downloaded plugin alongside the existing user interface software under the fan1 namespace of the home Meshpoint. Where the name of a control did refer to a control at another Meshpoint, such as the wb.temperature control of the Weather Bureau, the name never needed to change regardless of what Meshpoint the plugin was downloaded to.

Here, the Remote control from other fan plugin has to be told that the ‘other fan’ is fan2 by prefixing all its controls that refer to the other fan with the fan2 namespace. (To reflect that the plugin needs to be customised, the Remote control from other fan setting in the simulation—shown in Figure 3.30 on page 93—is Downloaded and customised, instead of just Downloaded.)

Figure 3.31 on page 95 shows the Remote control from other fan plugin.

3.14.5 Linked fan operation

The Remote control from other fan plugin maps changes that occur in the state of the two fan2 keys to the corresponding keys in fan1. Pressing or releasing a fan2 key has the same effect as if the equivalent fan1 key had been pressed or released. The fan1 keys do not physically move up and down when this happens, since, as their in direction spin hardware definitions indicate, the user interface cannot control what the keys do. It is just the abstract representation of these keys in the fan1 user interface that changes\(^5\).

Each press of a key on fan2 will now not only control fan2 operation in the usual way, but fan1 operation too. If both fans are running the same user interface, then fan1 will run exactly the same way as fan2. This doesn’t stop fan1 from being directly controlled from the fan1 keypad, but

\(^4\)The term superprogram takes its inspiration from superorganism, used to describe social insects such as ants and bees. The individual organisms do not work on their own, being so interdependent that they cannot survive by themselves. Do they form a colony of organisms, or is the organism the whole colony?

\(^5\)In practice, this plugin could lead to undefined behaviour since it is not clear what will happen when the plugin changes the abstract representations of keys, given that these representations are maintained by device firmware to reflect the actual state of the keys at all times. For example, what will happen if the plugin changes the whitekey control to Down while the corresponding white key is still physically up? Actual behaviour will depend on how the firmware has been implemented. A safer alternative would be to edit the user interface and change every procedure based on a particular key press in fan1, to one based on either that key or the equivalent key in fan2. However, the plugin is used here in the simulation since it allows greater freedom to experiment with different combinations of user features represented by the various other plugins and user interfaces.
if this happens, the two fans may get out of step with each other so that they no longer do the same things when controlled from fan2.

3.14.6 Interaction invited—not imposed

This example demonstrates the Mesh principle that the control of one program by another is invited rather than externally imposed.

To make it possible for fan2 to influence the running of fan1, code was added to the user interface of fan1—not fan2. This code provided the additional functionality of allowing fan2 to control its operation. It was not a case of adding code to the fan2 user interface that would force fan1 to work a certain way.

The normal convention in superprogramming is that programs are not allowed to directly change the setting of controls in other programs. If this is the intention of the relationship between two
programs, it is up to the other program to change the setting itself by making the change dependent on conditions in the first program. This approach ensures that coding integrity is maintained, since when a program runs, all changes to its control settings remain at all times a direct reflection on how its code has been written.

While this convention is clearly the safer option when a program is exposed to other programs running in the Mesh, there are times when it is useful for a program to accept externally made changes to its control settings. An example of this was demonstrated in the Panning timeout plugin where the ghost-in-the-machine API Meshpoint allowed its settings to be changed by client-side programs as a means of providing client-side support of method calling. But this, of course, was somewhat of a special case since the ghost-in-the-machine API Meshpoint acts as a unique Mesh resource, responding individually, not collectively, to each client-side program (a distinction that the client-side program does not need to know).

As we have already seen in the abstract hardware layer, the standard direction spin controls—in, out and inout—are used to specify the relationship between a control and its external world. The only difference in their use here is that the external world they refer to isn’t device hardware but software running at other Meshpoints. For a control to be exposed to externally made changes to its setting, it would need to be declared with in or inout spin.

### 3.14.7 Superprogramming implementation issues

The implementation issues raised by superprogramming fall into two broad categories: those concerned with how the programs run and those that have a direct impact on the Internet itself.

Some of the programming issues that every program must face, regardless of what language they are programmed in, will now apply on the grander scale of a network of Internet-linked programs. Infinite loop detection is a typical example. In the same way that a program can get caught in an unintended, infinite loop that stops it from performing its required task, interacting Meshoil programs may lead to similar, infinite loop problems extended across the Internet. Standard mechanisms for handling infinite looping in a program would need to be expanded to cope with the bigger picture.

The question for the Internet is, could it support the volume of Internet traffic that superprogramming would generate? Compared to the Web, we might expect a greater number of requests for information transfer, but a smaller amount of information transferred each time. In the Web, a single transfer typically consists of the information required to describe a complete Web page. In the Mesh, it would amount to just the setting of a single control.

Would the Internet be able to service the requests for information transfer between executing Meshoil programs fast enough in order to avoid a backlog of pending requests from building up? It is beyond the scope of this research project to answer these questions, but many issues might be resolved with minimal effort by employing caching mechanisms similar to those mentioned in section 3.11.4 on page 83.

As the infrastructure of the Internet continues to improve, any concerns with the volume of Internet traffic generated by the Mesh are likely to lessen. A measure of the current capacity of the Internet to handle high volumes of traffic is indicated by some of the Web applications it is used for. A typical example is Joost [22]—the online, multi-channel TV project created by the inventors of Skype for streaming near DVD-quality, full-screen video to millions of computers simultaneously.
3.15 “I just want to control the kitchen fan remotely”

Control settings for this simulation run:

![Simulation settings: fan2 as a hand-held remote controller](image)

3.15.1 Bedroom fan as a hand-held remote controller

Should the user want to use fan2 in the bedroom purely as a remote keypad for operating fan1, all he would need to do is to deactivate the fan2 user interface (equivalent to setting the fan2 USER INTERFACE control to ‘-’ in the simulation) while leaving fan1 unchanged with its user interface and Remote control from other fan plugin still working.

This would be another example of hardware reuse discussed earlier, since now the fan2 device would operate as if it was a simple, hand-held remote controller instead of a fan.

3.15.2 Point-accurate interoperation

This example is interesting for the way it highlights the difference between the philosophy of the Mesh and conventional engineering thinking. It focuses on the issue of remote control, something that is routinely engineered into many electronic products and appliances.

An often heard complaint is that our lives are snowed under with remote controllers, each one working differently for a different device, with no way to integrate their operation and replace them all with the convenience of a single unit. Using conventional engineering, there is no straightforward solution to this problem, but as this example shows, the problem is easily addressed in the Mesh through superprogramming.
CHAPTER 3. PRINCIPLES OF THE MESH

Consider the conventional engineering solution for controlling a device from a remote controller:

- the remote controller is dedicated to its task
- a command-based language is invented
- knowledge of how the device works is built into the remote controller in the form of a list of commands that can be transmitted to the device
- the extent to which the remote controller can control the device depends on how comprehensive the command set is in representing everything that the device can do
- the device is programmed to recognise commands from the remote controller and trigger the corresponding processes in the device.

The Mesh doesn’t follow this model. Interoperation is achieved by adding the necessary controlling logic to the device at the exact location in the code where interoperation is designed to have its effect. In line with OPT principles, this approach of point-accurate interoperation not only reduces the work to be done in making the system interoperable, but it also increases the flexibility of the results that can be achieved.

Take this example, where the user wants to use the bedroom fan in the manner of a remote controller by making its keypad work just as if it was the keypad on the kitchen fan. The only way to achieve this conventionally would be to implement a command set that represented every operating feature of the fan, and program the bedroom fan to mimic fan operation by issuing the right command for each key press of each user procedure. The Mesh, however, addresses the very heart of the issue—the user’s wish for equivalence of the keys—by directly mapping the operation of the bedroom keypad to the kitchen keypad.

The benefits of the Mesh approach over a conventional solution are immediate:

- programming effort is reduced
- no programming of the remote controller is necessary
- code duplication is avoided
- transmission overheads are reduced
- problems of scale are avoided
- the flexibility and scope of interoperation is maximised.

Programming effort is reduced because there is neither the need to create SIC constructs like command sets nor the need to program the remote controller and device to handle command transmission and interpretation. In fact, no programming of the remote controller is required at all, and the only programming needed in the device is minimal, localised changes to the code that controls how the device works.

There is never the sense of having to duplicate code, as typified by a command set that reflects and overlays basic device operation. Duplicating code is never a good thing since it inevitably introduces problems and inefficiencies. For example, changing how the device works in a command-based system may make the command set in the remote controller out of date and in need of updating to reflect the changes.

Transmission overheads are reduced because the size of the transmitted information is reduced. Instead of transmitting a command that has to convey sufficient information to indicate exactly
what the device has to do, the transmitted information is never more than the single value of a
programming variable (in the example here, the binary state of a control representing a key on the
keypad).

Problems of scale do not arise because the solution isn’t heavily engineered as in the conventional
approach. For example, should the remote controller be used to control a large number of different
devices simultaneously, the solution would be structurally no more complex. Compare this to
the challenge faced in a command-based system of programming multiple command sets into the
remote controller and making each key press transmit an increasingly long series of commands.

Finally, since the mechanism for achieving interoperation is based on the code controlling the
device itself, and not on some representative software overlay, every aspect of how the device
operates is amenable to interoperation by default without any effort being required.
3.16 “Give me bi-direction control”

Control settings for this simulation run:

![Figure 3.33: Simulation settings: Bi-directional remote control](image)

3.16.1 Bi-directional remote control

Should the user want bi-directional remote control so that both fans respond to key presses regardless of which keypad is used, he could repeat what he did for fan1 by installing the Remote control from other fan plugin in fan2 and customising it to recognise the fan1 keypad. Doing this would effectively make the two keypads one and the same piece of hardware.

Now, so long as both fans are running the same user interface and nothing else affects their operation, the two fans will operate in unison, regardless of which fan is used.

But only the two keypads have been made one. The fans themselves are still fundamentally running independently of each other. If anything upsets their parallel running, for example, they both use the Thermostat plugin for temperature control but with different thermostat settings, their operation can still get out of sync.
3.17  “I WANT THE FANS TO BE CLONES OF EACH OTHER”

Control settings for this simulation run:

```
<table>
<thead>
<tr>
<th>Fan 1 User Interface</th>
<th>Standard (11-speed, pan)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>- Convert 3-speed to 5-speed</td>
<td></td>
</tr>
<tr>
<td>- 1. Black key on/off</td>
<td></td>
</tr>
<tr>
<td>- 2. White key 6-speed</td>
<td></td>
</tr>
<tr>
<td>- 3. Black and white key panning</td>
<td></td>
</tr>
<tr>
<td>- 4. Thermostat</td>
<td></td>
</tr>
<tr>
<td>- 5. Panning timeout</td>
<td></td>
</tr>
<tr>
<td>- 6. Safety cutout</td>
<td></td>
</tr>
<tr>
<td>- Remote control (other fans)</td>
<td></td>
</tr>
<tr>
<td>- Clone of other fan operation</td>
<td></td>
</tr>
<tr>
<td>- Only usable if other fan is on</td>
<td></td>
</tr>
<tr>
<td>- Start panning if other fan stops panning</td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Fan 2 User Interface</th>
<th>Standard (11-speed, pan)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>- Convert 3-speed to 5-speed</td>
<td></td>
</tr>
<tr>
<td>- 1. Black key on/off</td>
<td></td>
</tr>
<tr>
<td>- 2. White key 6-speed</td>
<td></td>
</tr>
<tr>
<td>- 3. Black and white key panning</td>
<td></td>
</tr>
<tr>
<td>- 4. Thermostat</td>
<td></td>
</tr>
<tr>
<td>- 5. Panning timeout</td>
<td></td>
</tr>
<tr>
<td>- 6. Safety cutout</td>
<td></td>
</tr>
<tr>
<td>- Remote control (other fans)</td>
<td></td>
</tr>
<tr>
<td>- Clone of other fan operation</td>
<td></td>
</tr>
<tr>
<td>- Only usable if other fan is on</td>
<td></td>
</tr>
<tr>
<td>- Start panning if other fan stops panning</td>
<td></td>
</tr>
</tbody>
</table>

3.17.1 Cloned fans

The user decides he doesn’t just want remote control, he wants the two fans to be true clones of each other, identical in every respect just as if they were physically one and the same unit.

He downloads the Clone of other fan operation plugin from the ui4u Meshpoint (shown in Figure 3.35 on page 102), customises it in the usual way by making fan1 the other fan, and installs it in the fan2 user interface in place of all the existing software. He leaves fan1 unchanged from the previous example of bi-directional control, running the standard user interface and the Remote control from other fan plugin.

Now the two fans never get out of sync, regardless of what happens.

3.17.2 Reciprocal mapping of hardware controls

The motor and pan controls determine what each fan is doing at any one time. To make the two fans work the same way, the Clone of other fan operation plugin maps the state of fan1’s motor and pan controls to those of fan2—fan2 running becomes a clone of fan1 running.

fan2 has no need of a user interface of its own, since it relies totally on what fan1 is doing. Its operating heart is just a mirror of what is happening in fan1. Should fan1 switch to the use of a different user interface, it will immediately seem to the user that the user interface has changed in fan2 as well.

The keypad on fan2 is still used to control fan1 remotely, so to complete the cloning arrangement fan1 still needs to be set up with its Remote control from other fan plugin.

The same cloning result could equally well have been achieved by reversing the roles for the fans, having the user interface running in fan2, mapping the motor and pan controls of fan2 to those of fan1, and making fan2 respond to key presses on fan1.
Cloning means making identical copies of something. Notice that it could not be achieved by mapping the motor and pan controls of both fans to each other, since this might cause the fans to lock up. Their behaviour would become undefined, dependent on how the system was implemented, since any unilateral attempt to change the running of one fan would immediately break the rule that it has to be a mirror image of how the other fan is running at all times.

### 3.17.3 Multiple cloning

If more than two fans were to be cloned, a star network would be followed. The fan at the centre would be equivalent to fan1 in this example, running the user interface and responding to key presses on all the other fans. Each satellite fan would be equivalent to fan2 in this example, having no user interface of its own and mapping its motor and pan controls to those of the fan at the center. At any time, it would be possible to update how all the fans work by replacing the user interface in the one fan at the center.

An alternative approach would be for every fan to run its own user interface, making every procedure in the user interface able to be triggered not just by the key belonging to the particular fan, but by the equivalent key on any fan.
3.18 “I want to control my son’s use of the fan”

Control settings for this simulation run:

<table>
<thead>
<tr>
<th>Fan 1 User Interface</th>
<th>Fan 2 User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert 3-speed to 5-speed</td>
<td>Convert 3-speed to 5-speed</td>
</tr>
<tr>
<td>1. Black key on/off</td>
<td>2. White key 5-speed</td>
</tr>
<tr>
<td>3. Black and white key panning</td>
<td>4. Thermostat</td>
</tr>
<tr>
<td>5. Blower timeout</td>
<td>6. Safety cutoff</td>
</tr>
<tr>
<td>Remote control from other fan</td>
<td></td>
</tr>
<tr>
<td>Clone of other fan operation</td>
<td></td>
</tr>
<tr>
<td>Only usable if other fan is on</td>
<td></td>
</tr>
<tr>
<td>Start panning if other fan stops panning</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.36: Simulation settings: Restricted fan usability

3.18.1 Restricted fan usability

Our user has a young son who sleeps in the bedroom where fan2 is. He doesn’t mind his son using the fan on hot nights when he himself has the kitchen fan running, but he wants the peace of mind of knowing that his son won’t be playing around with the fan at times when he shouldn’t. He wants a system that will allow the bedroom fan to be used normally while his fan is on, but will ensure that the fan cannot be used at all other times.

He finds the Only usable if other fan is on plugin at the ui4u Meshpoint that does what he wants, so he downloads and installs it in fan2. Figure 3.37 on page 104 shows this plugin.

3.18.2 More subtle superprogramming relationships

Any level of relationship can be set up between executing Meshoil programs.

This example shows the more subtle effect of setting up a parent-child relationship between the two fans. While fan1 is running, fan2 is allowed to operate normally without influence from fan1. But as soon as fan1 switches off, control is effectively imposed on fan2 and it switches off too. fan2 will then remain disabled for as long as fan1 remains off.
CHAPTER 3. PRINCIPLES OF THE MESH

Figure 3.37: Only usable if other fan is on plugin
3.19 “I want the fans to take it in turns to pan”

Control settings for this simulation run:

```
3.19.1 Alternate panning
Our days that are a bit stuffy but not terribly hot, the user wants panning to run intermittently, rather like the action of a car’s intermittent windscreen wiper used when the rain is spitting. He wants panning to alternate between the two fans, so that each fan takes it in turn to pan for a while.

He finds the Start panning if other fan stops panning plugin and realises that he can get what he wants by combining its use with the Panning timeout plugin he has used before (section 3.9.1 on page 70). By installing both plugins in both fans, he will be able to get the fans to alternate panning.

Figure 3.39 on page 106 shows the Start panning if other fan stops panning plugin.

3.19.2 Control feedback loop
In the Start panning if other fan stops panning plugin, the event of panning stopping in one fan is the trigger to start panning in the other. The control feedback loop is completed by the Panning timeout plugin that automatically stops a fan from panning after a certain time.

The logic in these two plugins ensures that once the user has switched the panning timeout feature on in each fan by specifying a timeout period, and once panning has started in either fan—for whatever reason, whether by the user pressing a key or some other automated procedure—the feedback loop will start and continue to run. Panning will then constantly alternate between the two fans until the loop is finally broken by one of the fans being switched off.
3.19.3 Cascading interactions across the Mesh

In the examples of superprogramming discussed so far, interaction has been limited to simple cause and effect. An event happening in one fan has triggered something to happen in the other fan. But in this example, the story doesn’t end there because the event triggered in the second fan is itself the trigger for something else happening. A sequence of events occur rather than just one.

In superprogramming, not only can one program trigger any number of events in any number of other programs, but each of these events may subsequently trigger further events in other programs, creating a cascading effect that travels across the Mesh as a wave of ongoing, software-driven activity.

While there is no limit to what a single event in one program can ultimately lead to, these patterns of behaviour are not random but highly deterministic, the sum result of the chosen, individually programmed operations of all the players involved.
3.20 “I want the fan to run more efficiently”

Control settings for this simulation run:

![Simulation settings: Advanced (5-speed, pan) ghost processor user interface](image)

The user interface only runs while the user’s personal computer is on, simulated by the `On` setting of the `mesh://mycomputer.power` control at the bottom of the main control window shown in Figure 3.1 on page 35.

3.20.1 Increasing executional efficiency of a program

Worried that the processor built into the fan for running the user interface is a bit underpowered for the increasingly sophisticated user interface designs he is running, the user considers various ways to make the system run more efficiently.

Taking the advanced 5-speed user interface program as an example, he replaces it with a multithreaded version that runs more efficiently. This helps, but doesn’t overcome the basic problem of the limited processing power of the fan’s in-built processor.

Then he gets the idea of outsourcing the job of running the user interface to another device altogether. He uses his personal computer as the other device, since it has a more powerful processor. This new arrangement significantly reduces the workload for the fan’s processor.

Happy with the increased efficiency and reliability of operation that this solution brings, the user decides to use his computer increasingly this way as a handy server for running many of the devices and appliances around his home.

3.20.2 Multithreading a program

Figure 3.41 on page 108 shows the outline of the program for the original **Advanced (5-speed, pan)** user interface for fan1. The program consists of two bricks: a `with` brick defining the hardware controls, and a `do` brick defining how the fan works according to the settings of the hardware controls. The program runs as a single thread.

This arrangement is common to all the user interface examples discussed so far in this chapter.
Figure 3.41: Program running as a single thread

Figure 3.42 shows how the same program has been made to run more efficiently by multithreading.

Figure 3.42: Program multithreaded for increased executional efficiency
In Meshoil, the code within a do brick runs as a separate thread (see appendix A.4.4 on page 304). Here, the user interface code has been broken up into three threads that independently handle Black key operations, White key operations and Black and white shift key operations (for a more detailed example of multithreading, see section 5.1.8 on page 141). The natural modularity of a Meshoil program can be exploited this way to break up the program into any number of threads.

In the concept demonstrator, all threads run on a single processor as time-sliced tasks, but on a multiprocessor or multi-core system, the multiple threads of a Meshoil program could literally run concurrently.

### 3.20.3 Grid computing

So far, the examples in this chapter have discussed the various ways that information about controls can be obtained from remote locations in the Mesh and incorporated into the running of a Meshoil program. In each of these examples, everything to do with the execution of the program has remained local to the device where the program runs. However, in the same way that a program can outsource the task of managing control settings, it can outsource the very task of processing them too. Thus, dynamic as well as static aspects of a program can be handled remotely.

This capability demonstrates how the Mesh supports the concept of Grid computing, where program threads run simultaneously on a number of disparate, geographically distributed devices communicating with each other over a network.

Grid computing is one of several models for supporting concurrent processing on a large scale—two others being cluster computing and distributed computing. Grid computing has the most generalised application, since unlike other models, there is no requirement for the processors to be similar in nature or physically located close to one another. However, this ability comes at the cost of only being able to support limited communication and sharing of information between the concurrently executing threads.

Conventional grid computing works best for large problems that are able to be broken down into many smaller, individual ones, each one worked on in relative isolation by a different processor before being collated and reassembled into creating the solution to the original problem [40].

### 3.20.4 Ghost processing

In the Mesh version of grid computing, it is possible for threads to communicate and share information with one another at any time while they are concurrently executing, just as if they were executing on the same programming platform. The approach is more akin to parallel processing supported within a conventional programming language, but expanded from a single platform to embrace a geographically distributed and disparate collection of devices having in-built processors.

The need for heavily engineered, conventional methods and protocols for supporting grid computing—that still only manage to achieve limited information sharing—is avoided. The solution drops naturally out of the design of the Mesh and the fact that every control in the Mesh is:

- uniquely identifiable by its name and Meshpointer
- universally accessible.

Maintaining a close physical connection between where information is held and where it is processed is no longer necessary. The link between the two can be freely broken as convenience dictates. The freedom to do this has already been demonstrated in this chapter (eg, examples in
sections 3.8.1 and 3.14.1 on pages 64 and 93). The only difference here is that all controls, rather than just some, are remotely accessed.

Figure 3.43 builds on the previous example of a multithreaded program to demonstrate the various ways that the Mesh supports grid computing. The job of running all three threads of the user interface is outsourced so that the only remaining task of the fan’s in-built processor is to manage the interface with the fan’s hardware controls, consisting of the keys and other physical features of the device.

![Ghost processing diagram](Image)

**Figure 3.43: Ghost processing**

The `do` brick for the thread that controls Black key operations is declared as `do(mesh://my computer)`. By default, a `do` brick executes in the local device, but here it has been qualified with the `mycomputer` Meshpointer of the user’s personal computer—a Meshable device just like the
fan itself. This makes the thread run in the computer instead. Whenever execution of the user interface program starts in fan1, mycomputer is called upon to act as an independent server—a virtual processor—for running this thread. This effect of boosting the computing power of the fan1 device through outsourcing is called ghost processing.

Processors in all Meshable devices would be designed to support ghost processing. Included in their operating systems would be transparent mechanisms for responding to and handling client-side requests for running server-side threads. On receiving such a request, the operating system of the processor would determine whether to accept or refuse the request depending on factors such as current CPU loading. While all Meshable devices would offer this functionality, it would be possible for it to be disabled for simple Meshable devices that did not want to be used in the bigger picture of grid computing applications.

Server-side threads would run as self-contained, background applications, independent of resident, client-side programs and any server-side threads of other grid computing applications. Malicious software problems would not arise because server-side threads would effectively be firewalled with no means of directly interacting with the hardware or software resources of the device.

Concurrent running of client-side and server-side code in a device is demonstrated in the do brick for the second thread that controls white key operations. Declared as do(mesh://fan2), this part of the fan1 user interface runs in the fan2 device alongside the resident user interface for controlling fan2.

In many ghost processing situations, it might not matter where a thread was outsourced to. This is demonstrated by the third thread that controls black and white shift key operations, where the do brick is declared as do(mesh://grid).

grid is a special Meshpoint that acts as the grid service manager for the Mesh, taking on the responsibility of choosing which processors to use. It handles such issues as processor allocation from all those currently available in the Mesh, and switching to an alternative processor should the one allocated suddenly become unavailable for any reason.

### 3.20.5 Remote program management

In the previous example, although all software for running the user interface was outsourced, program management was still performed locally in the device. In this example, however, even this aspect of the fan’s user interface has been outsourced.

Figure 3.44 on page 112 shows how the code for the three threads has been moved from fan1 to mycomputer. Management of the user interface program, such as initiating its execution, is now performed directly from mycomputer.

From the program’s point of view, the local device is now mycomputer instead of fan1, since this is where the executable part of the program resides. Accordingly, the do bricks for the threads are simply declared as ‘do’ once again in order to make them run in mycomputer. All fan1 hardware controls referenced in the code of the three threads need to be specified with their full Meshpointers, since these controls now have to be accessed over the Mesh. For example, instances of blackkey are replaced by mesh://fan1.blackkey.
Complete code outsourcing

The final step in outsourcing the user interface is to outsource even the definition of the fan’s hardware controls. Figure 3.45 on page 113 shows how everything to do with the definition of the user interface has been moved from fan1 to mycomputer, leaving fan1 empty of all code and available for even more flexible patterns of usage.

In mycomputer, the with brick containing the definition of hardware controls is declared as with(mesh://fan1). This is the part of the program that sets up the transparent interface between the physical controls of the fan and their equivalent software representations accessible from within a Meshoil program. When execution of the user interface is initiated from mycomputer, this definition is downloaded to fan1 before execution of the user interface begins.
3.20. "I WANT THE FAN TO RUN MORE EFFICIENTLY"

```
<table>
<thead>
<tr>
<th>Complete code outsourcing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>... mesh://fan1</td>
</tr>
<tr>
<td>fan1 Meshpoint</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>User Interface</td>
</tr>
<tr>
<td>Empty of all code</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>... mesh://mycomputer</td>
</tr>
<tr>
<td>mycomputer Meshpoint</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>User Interface</td>
</tr>
<tr>
<td>Everything to do with</td>
</tr>
<tr>
<td>specifying the fan1 user</td>
</tr>
<tr>
<td>interface is handled</td>
</tr>
<tr>
<td>remotely in mycomputer.</td>
</tr>
<tr>
<td>This includes the</td>
</tr>
<tr>
<td>definition of the</td>
</tr>
<tr>
<td>fan1 hardware controls</td>
</tr>
<tr>
<td>as well as their usage</td>
</tr>
<tr>
<td>in user interface</td>
</tr>
<tr>
<td>operation.</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>(v)</td>
</tr>
<tr>
<td>Hardware controls</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>(v)</td>
</tr>
<tr>
<td>Black key operations</td>
</tr>
<tr>
<td>Controls specified with</td>
</tr>
<tr>
<td>full Meshpointers.</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>(v)</td>
</tr>
<tr>
<td>White key operations</td>
</tr>
<tr>
<td>Controls specified with</td>
</tr>
<tr>
<td>full Meshpointers.</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>(v)</td>
</tr>
<tr>
<td>Black and white shift key</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>Controls specified with</td>
</tr>
<tr>
<td>full Meshpointers.</td>
</tr>
</tbody>
</table>
```

Figure 3.45: Complete code outsourcing
CHAPTER 3. PRINCIPLES OF THE MESH

3.21 “I want software that I can trust”

Control settings for this simulation run:

![Figure 3.46: Simulation settings: Faulty advanced (3-speed, pan) user interface]

In this simulation, the faulty version of the Advanced (3-speed, pan) user interface is run in the user’s personal computer to see how it performs before being considered for installing in the fan. The personal computer must be on, simulated by the On setting of the mesh://mycomputer.power control at the bottom of the main control window shown in Figure 3.1 on page 35. Diagnostics display in mesh://mycomputer.display as faults are detected during operation of the user interface.

3.21.1 Faulty advanced (3-speed, pan) user interface

Whenever the user downloads software from the Mesh, he wants to know that it is safe and reliable before installing it in the fan and using it to control fan operation.

He has faith in software provided at the ui4u Meshpoint because he knows that all their software undergoes reliability testing as part of ui4u’s software accreditation policy. He is also reassured by their user feedback scheme on how well their software performs and the incorporation of this feedback into the accreditation process. ui4u jealously guard their reputation as a reliable Meshoil software provider because they know their business depends upon it. They have managed to built up a loyal customer base of users who have learnt to trust their products.

This time, however, the user is lured by a Meshpoint called resoft, known for recycling old user interfaces as free shareware. He finds a version of the Advanced (3-speed, pan) user interface that he is tempted to use. resoft’s motto is strictly ‘caveat emptor’, since they make no claims to the reliability of their software.

With none of the standard checks in place offered by the ui4u Meshpoint for ensuring code integrity, the user takes measures of his own. He decides to test out how well the user interface works on his personal computer before going live with it in his fan. He uses a software application called the Meshoil checker that subjects Meshoil programs to various tests for the purpose of establishing code integrity and reliability. All he has to do is to download the user interface to his computer and run it through an ‘oil-check’.

Rather than run the battery of automated tests, he selects manual mode in the Meshoil checker and chooses to operate the user interface himself from a GUI that simulates the working fan. He finds that everything works OK except for panning. Instead of the panning setting toggling just once each time the black key is pressed with the white key as shift key, the setting chatters between On and Off for as long as he keeps his finger on the black key. Clearly, this part of the program has been badly written.

His first thought is to give up on using this user interface since he is not a programmer and...
wouldn’t know how to fix the problem, but then he realises that the Meshoil checker gives him a better alternative. He can use the Meshoil checker to filter out just the bad parts of the program. Although the result will be less than perfect, since he won’t be able to control panning, at least he will be left with a working, reliable user interface capable of switching the fan on and off as well as controlling the speed—better than nothing for something that was free!

3.21.2 Code filtering

When a conventionally written program contains errors, there is no easy way of managing the problem without the help of a programmer able to debug the code. The natural modularity of Meshoil and support for cut and paste programming, however, mean that code filtering is possible: sections of code can easily be identified and removed while still leaving a viable program capable of being compiled and executed. Bad areas of a program can be filtered out this way to leave the remaining, working parts intact and functioning.

Figure 3.47 on page 116 shows the part of the user interface in this example responsible for the black key timing problem. The rise control was inadvertently missed out from the definition of the mesh://fan1.blackkey—Down condition (the correct code for the Advanced (3-speed, pan) user interface is shown in section 3.5.1 on page 50). The user didn’t have the necessary skills to understand the nature of the problem and fix it himself, but he was still able to come up with a better solution than simply rejecting the whole program. Having been alerted to the problem by the Meshoil checker, he followed its prompts for deleting the section of offending code from the program.

3.21.3 Code filtering as a tool for customising software

As well as a tool for ensuring software reliability through the removal of bad code, code filtering can be used on good code as a way of customising a program.

Section 3.7.1 on page 60 discussed the plug-and-play design approach for customisation in which software is progressively added to a program. Customising a program by code filtering takes the reverse approach of progressively removing software. A program starts out with a comprehensive set of features designed to meet the needs of a range of users. Tailoring the program to suit a particular user then involves removing the unwanted features.

While more restricted in scope as a method for customisation—since it is never possible to go beyond what the original program is capable of—code filtering might suit situations where limits need to be set for the different ways that a program can be customised. A typical example might be a device manufacturer who wants to exercise control on what their products can be made to do.

3.21.4 The problem of unreliable and malicious software

The freedom that the Mesh brings for sharing software over the Internet and changing how devices work inevitably raises the problem of unreliable and malicious software. The issue is particularly important since Meshoil programs would be used to control the physical operation of devices and machinery. In some situations, incorrect control, whether intentional or not, could risk harm to device or user. The principle of user-generated software would also encourage those with little or no programming skill to start producing software of their own.
The design of the Mesh would work to limit the scope for malicious software since:

- Meshoil programs run in devices as tightly firewalled software applications (section 3.20.4 on page 109)
- there is no easy way to access the OS and hardware resources of a device from a Meshoil program
- control of one program by another has to be expressly invited (section 3.14.6 on page 95).

3.21.5 Code integrity through accreditation

The Web serves as a useful model for how the Mesh would combat the problem of unreliable and malicious software. Many of the strategies for maintaining the integrity of user-generated content
in the Web would have direct parallels in the Mesh, the only difference being in the detail where content refers to software instead of media.

The Mesh would ensure *code integrity through accreditation*, involving a mix of manual and automated methods—some computer-based, some socially driven. Safeguards against bad software would be based on a strategic approach that would include:

- manual accreditation
- automated accreditation
- social accreditation.

### 3.21.6 Manual accreditation

Websites often filter uploaded content in order to guard against both factual errors and information seeking to cause mischief through being misleading or profane. Offending material is removed as it is encountered, or if the extent of the problem is deemed too severe, the content is rejected completely. A typical example is the Wikipedia [51] website. Wikipedia’s editorial team constantly checks uploaded content to ensure that it remains accurate and unbiased, a necessary task for maintaining the quality and reputation of the website.

Use of the Meshoil checker in this example shows how the Mesh is able to parallel the filtering of content practised in the Web. The honest mistake of factual inaccuracies and the intentional nature of misleading information in knowledge-IT equate to the twin problems of unreliable and malicious software in control-IT. Owners of Meshpoints like ui4u would follow the Wikipedia model for ensuring content integrity. This might involve manual inspection of uploaded software as well as performance testing using tools like the Meshoil checker. Editorial discretion would be used to categorize the software products uploaded to the Meshpoint according to factors such as reliability, usefulness for different types of user and ease of operation.

A particular advantage of using the Meshoil checker is that it allows a Meshoil program such as a user interface to be run in abstract form, isolated from all device hardware. Hardware controls are simulated in a GUI, much like the concept demonstrator itself, and their physical links to items of hardware via the abstract hardware layer are simply ignored. This makes the software accreditation process both convenient and safe, since testing can be performed anywhere without risk of causing damage to the device.

### 3.21.7 Automated accreditation

As well as manual inspection, the Web relies heavily on automated filtering of content. The same would be true in the Mesh where much of the software accreditation process would also be automated.

Automated accreditation is made possible by the fact that much of how a program is designed to control a device is known from the nature of the hardware controls and how they are used in the program. This knowledge is assisted by the Mesh principle of globally-unified hardware control (section 3.10.6 on page 80) where the names of hardware controls—such as power, key and motor—are standardised to give them recognisable meaning in their role as counterparts to the real world.

For example, it would be known that the start of a user procedure must be triggered by a change in state of a hardware control such as a key that had in direction spin, since only such controls provide the means for the user’s physical interaction with the device. Automated accreditation would search for such inferred code signatures in the program and experimentally run the program...
CHAPTER 3. PRINCIPLES OF THE MESH

by changing the states of controls to simulate manual operation. The effect of setting different combinations of control states would be tested, akin to the tedious task of exhaustively testing device operation by manually operating every physical means of control for each situation in which the device is intended to be used.

Table 3.1 gives examples of the type of programming problems that the Meshoil checker would be able to identify when running in automated accreditation mode.

<table>
<thead>
<tr>
<th>Identified problem</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potentially harmful proce-</td>
<td>The program causes a motor control to oscillate or rapidly switch between speed settings that might be physically damaging to the corresponding motor in the device.</td>
</tr>
<tr>
<td>dure</td>
<td></td>
</tr>
<tr>
<td>Incomplete user procedure</td>
<td>The program contains code for setting the power control On but none for setting it Off, indicating that the user will have no way of switching the device off once it has been switched on.</td>
</tr>
<tr>
<td>Impossible operation</td>
<td>The program attempts to change a key control to its Down setting, something that only the user can do by physically pressing the corresponding key on the device.</td>
</tr>
<tr>
<td>Hardware feature unused</td>
<td>Output as a caution: the program makes no apparent use of an available hardware control.</td>
</tr>
<tr>
<td>Conflicting operation</td>
<td>Different procedures triggered by the same change in the setting of a hardware control (eg, a key) lead to conflicting settings in other controls.</td>
</tr>
<tr>
<td>Inefficient code</td>
<td>A procedure takes more steps than necessary to achieve a given outcome (see example in section 5.1.7 on page 140).</td>
</tr>
<tr>
<td>Useless code</td>
<td>The program either contains code that never executes or a procedure that has no apparent effect on how the device is working.</td>
</tr>
</tbody>
</table>

Table 3.1: Examples of problems identified in automated accreditation of software

3.21.8 Social accreditation

In addition to the use of tools such as the Meshoil checker, software accreditation would be supported by other, non computer-based methods. As for any type of commercial product, software providers might choose to reassure customers in the reliability of their software by providing guarantees. The Web, however, shows how the interconnectivity of the Internet is opening the door to new, socially driven methods for ensuring reliability.

A typical example is eBay’s [10] peer review system. eBay members gain a reputation from their trading on the website that serves as a useful indicator of the extent to which they and the products they sell can be trusted. This reputation is built up from feedback received from customers, and tabulated in the form of numerical scoring and posted comments.

The advantage of harnessing social forces demonstrated by the eBay example is that the system becomes largely self-policing. Less effort is needed by the owners of the website in maintaining the integrity of their enterprise, since this task is evenly shared and willingly undertaken by the website’s many users.

The Mesh would follow a similar model. Meshpoints such as ui4u would allow their users to rate the reliability of the software uploaded by the various software providers. Subjected to this social pressure, software providers would do their best to deliver safe, secure and trouble-free software knowing that their reputation and success, commercial or otherwise, depended upon it.

118
Chapter 4

Universal usability

4.1 The state of play in this research area

4.1.1 New methodologies needed

There has been considerable interest in applying universal usability principles to user interface
design. While the benefits to the user of such an approach are well known, such as easy personal-
alisation and a common way of operating a wide variety of devices, achieving the goal has proved
frustrating.

Fundamental usability principles are well understood, such as the importance of making the
user interface as simple as possible, consistency, and allowing the user to explore and recover from
mistakes. But even factors such as aesthetics are now known to affect usability: if a system feels
two nice to operate, we may tolerate more of its actual weaknesses [33]. Every user will have a different
set of needs. Even individual needs will change according to situation or over time as we age [29].

There is general agreement that no single implementation of a user interface can ever hope
to satisfy all [49]. What is required are user interfaces that can more easily be customised and
personalised to suit any user situation [1] [16] [35] [37]. New methodologies are needed to reduce
system complexity while expanding functionality [23] [38].

4.1.2 Flexibility seen as the key

The challenge is seen as finding a design approach that can accommodate a dynamic and diverse
range of user requirements. User interfaces that simply support adaptability are not good enough.
They must be easy to adapt too.

The key to everything can be summed up in a single word: flexibility.

4.1.3 Slow progress to date

Various attempts have been made to create a universal user interface. Law’s [27] approach was to
offer users an alternative, consistent way of scrolling through and accessing any standard device
function using three additional keys (‘EZ Access’ keys)—the keys work in parallel with the existing
interface. Akoumianakis [2] developed a declarative, user interface development tool designed to
help the interface designer to create customised user interfaces. Shneiderman [39] promoted the
idea of multi-level user interfaces, designed to target a number of different categories of users such
as novices and experts.
Attempts like these do achieve some measure of success, but they fail in their bid to be truly universal solutions to the user interface problem since none of them support sufficient design flexibility to meet the individual needs of a user.

4.1.4 The XML trend

A more recent trend has been the switch to the use of XML-based languages for describing user interface design. Many of these User Interface Description Languages (UIDLs) hark back to their XML mother tongue with names like:

- UIML (User Interface Markup Language) [48]
- UsiXML (USer Interface eXtensible Markup Language) [47]
- XForms (XML Format Web Forms) [53]
- XIML (EXtensible Interface Markup Language) [11]
- XUL (eXtensible User Interface Language) [12].

There is considerable debate on which UIDL is the best. Trewin, Zimmermann and Vanderheiden [43] list the following six desirable features of a UIDL:

- applicable to any target (a target can be a physical device such as a TV or a service such as an airline reservation system)
- applicable to any context of use
- personalisable
- flexible
- extensible
- simple.

The first five points are really one and the same argument for maximum inherent flexibility in the language so that it can meet as many differing needs as possible—whether that means handling different devices, user needs or future requirements. The last point, simplicity, says that all this flexibility should be achieved by the simplest means possible.

With user applications becoming increasingly integrated, trying to maintain the same user interface rendered a number of ways on different platforms is becoming increasingly problematic and wasteful of resources. The answer is seen as having a single, abstract, device-independent user interface description that can easily plug into these platforms with minimal fuss. In support of this goal, UIDLs are essentially attempts at providing the means to repackage a user interface for different situations and modalities.
4.1.5 Can UIDLs do the job?

Expressing design information in the text-based format of an XML file is easier than in the code of a conventional language. All things being equal, it is always better to use a simpler tool in place of a complex one. Simple languages have many advantages over their more complex counterparts, but the question is, can they still perform the tasks required of them?

Figure 4.1: Simple languages have many advantages
CHAPTER 4. UNIVERSAL USABILITY

4.2 Limitations of the UIDL approach

4.2.1 Usefulness lost

From an OPT perspective, this use of a simpler tool is a step in the right direction. But UIDLs suffer by being considerably more SIC than the conventional languages they seek to replace.

A conventional language has an abstract syntax. While the tool may be complex to use and require the specialised skills of a programmer, nothing about the use of the language for any particular application is built into the syntax—you can use Java just as well for modelling the orbits of the planets as you can for controlling a microwave oven. No constructs of the syntax predispose the language for use in one type of application over another. The language remains abstract and application-neutral.

This is no longer the case in UIDLs, where the independence between form and function is lost. UIDL syntax is designed to reflect the dedicated application of a ‘universal user interface’ by the particular choice and arrangement of the XML elements that make up the language. These XML elements attempt to define the key aspects involved in user interface design. Each UIDL has a different interpretation of what is important. By hard-wiring decisions about design in this way, the usefulness of UIDLs as universal tools is reduced.

4.2.2 Imposing, not handling, design structure

In UIML, a user interface is defined as consisting of six component parts: structure/flow, presentation style, content, behaviour, logic and presentation. UsiXML breaks down a user interface into groupings such as widgets, controls, containers, modalities and interaction techniques. In XIML, there are just five parts to a user interface: task, domain, user, dialog and presentation.

Which approach is correct? There are merits and disadvantages to all of them, and we find ourselves facing the familiar trade-off between simplicity and flexibility.

Each UIDL has a particular focus that adds bias and reduces its overall usability. For example, XForms and XUL are specifically designed for rendering a user interface as a Web-based application. As a result of all these limitations, UIDLs become vehicles for imposing, rather than handling, design structure. They can’t be expected to perform well unless they are used to solve problems that happen to match their particular strengths and none of their weaknesses.

4.2.3 Still dependent on a conventional imperative language

UIDLs do not attempt to solve the whole universal usability problem. They take a top-down approach in which the user interface is split into top and bottom ends [7]. The UIDL handles the top end as a broad brush, overview of device operation. The bottom end of the system that specifies the bulk of how the user interface works is still hard-coded in a conventional, imperative programming language. The UIDL functions as an organiser for assembling large, hard-wired pieces of the user interface puzzle, typically by issuing commands that the underlying conventional language executes to make the device work\(^1\).

The ability to customise the system is limited to controlling what commands are available and when, since for the most part the user interface is still heavily reliant on conventionally written code that remains inaccessible and SIC.

\(^1\)This approach is true for all current examples of markup languages, not just UIDLs. A particular language may contain what is called ‘procedural markup’, but rather than inferring a native capability for imperative programming this merely means embedded descriptions of actions that an underlying, imperative language must perform. The universal preoccupation of such procedural markup is the presentation of text—the TeX typesetting language being a typical example—and the supporting, imperative application is usually highly specialised for its task.
4.2. LIMITATIONS OF THE UIDL APPROACH

4.2.4 The false declarative/imperative divide

UIDLs pride themselves on being declarative, not imperative. Their aim is to state what happens but not how it happens, since this is left to the underlying hard-coded software to handle. An often cited paper by Landin [26] makes the point that there is no hard and fast distinction between declarative and imperative programming; a declarative program can be seen as a subset of an imperative one, and the difference between the two is just a matter of scale. So is such a clear cut declarative/imperative distinction useful, or even meaningful?

Consider the definition of a user interface for a phone. Taking a top-down, UIDL-like approach, we might start with: *It makes calls*. Since we haven’t said how it makes calls, our user interface would appear to be declarative so far. Then we add detail to say that: *It inputs numbers to make calls*. But how does it input number? *Buttons on the keypad are pressed in sequence to input numbers to make calls*. But how are these number sequences handled? *Buttons on the keypad are pressed in sequence to input numbers that are buffered until the Call button is pressed to make calls*. Already we find that our user interface description has acquired a considerable amount of procedural detail. We continue adding detail this way until eventually we have a fully working phone, without ever being aware of having crossed some great declarative/imperative divide.

It is perhaps more helpful to see the process of design as one of gradual transition—an increasing granularity of description. Starting from a highly simplified, terse declaration of what a user interface is meant to do, more and more detail is added until finally a working version is reached.

4.2.5 Offering only half of the solution

A purely declarative language can never provide the complete solution, since by its very definition it is not Turing-complete\(^2\)—it lacks the means to make things actually happen. At some point in the creation of a working user interface, imperative programming is needed. With the lack of any feasible alternative, this currently means writing at least some of the user interface description in a conventional, imperative programming language, with all the attendant problems of solution complexity and platform-dependence that this introduces.

UIDLs only focus on solving part of the problem. They are simpler programming tools to use and they do help to decrease platform dependence, but by remaining resolutely declarative, they forego the descriptive power to replace much of what the underlying, imperative language has to do. It is, perhaps, in recognition of this inherent weakness, that some UIDLs try to extend their range by supporting some conditional expression functionality. But such functionality is very limited in what it can be used for because it is embedded in a rigid, declarative framework.

4.2.6 A mistaken conclusion?

The author believes that in the UIDL approach, the trouble stems from viewing imperative programming as something that only a conventional, mainstream programming language can support. This leads to the false conclusion that if you want to avoid having to use a complex language, your only option is to keep user interface descriptions purely declarative because this is all that a simple language can ever hope to handle.

\(^2\)A programming language is Turing-complete if it is theoretically capable of performing any computational task, defined as being the task executed on a simplified model of a programmable computer known as a universal Turing machine. Languages that support general mechanisms for storing and changing values, logical branching and iteration are Turing-complete. Turing-completeness, in itself, says nothing about whether such general computation is easy to do. Such issues of style are more subtle and tricky.
CHAPTER 4. UNIVERSAL USABILITY

What this really shows is that a simple but highly expressive language is needed, one that can be used to work right across the declarative/imperative spectrum. That way, one simple tool can be used to do the complete job of describing and running a user interface.
4.3 **Meshoil as a language for user interface design**

### 4.3.1 The Meshoil alternative

Although Meshoil is an XML-based markup language, it is not a UIDL. Rather, it is an abstract, general-purpose, programming language capable of spanning the full declarative/imperative spectrum and supporting the complete process of building a user interface.

The declarative parts of a Meshoil program are written in bricks that have `with` and `withmethod` `spin`. Such bricks are not only concerned with low-level, housekeeping tasks such as programming variable definition, but also in setting up major relationships such as how a user interface program relates to the world in terms of the hardware that it controls and the platform on which it runs.

The imperative parts of the program are written in `do` bricks. The uniform syntax for both means that conventionally viewed declarative and imperative programming tasks can be freely intermixed as required.

Meshoil programs can still be physically structured into top and bottom halves to emulate the UIDL approach if this is desired (for an example, see section 5.6 on page 202), but there is never a need to program in any other language.

The ability of Meshoil to emulate features of the various UIDLs is briefly discussed below. The list is not exhaustive, since it was beyond the scope of this research project to work through each UIDL in detail to consider how to match each feature, but it is at least indicative of how Meshoil has the potential to address the particular area of design that is the focus of each UIDL. Considerably more work into the nature of the various UIDLs would need to be done before fair and just comparisons could be made.

### 4.3.2 UIML

A recognised shortcoming of UIML is that while it supports the creation of user interfaces that can run on different platforms, there is no way to retain a core, abstract description of user interface function that is universal to them all [42]. Method calls for making use of the API provided by a particular platform are hard-coded into the body of the user interface description. As a result, UIML’s effectiveness as a universal language is reduced since different versions of a user interface description have to be created for running on each platform.

By effectively containing all such interaction between the user interface description and the API of the supporting platform at the single, clearly defined location of the abstract API layer, Meshoil avoids this limitation and is able to maintain an abstract core description of user interface function (see appendix A.5 on page 307).

### 4.3.3 XIML

One of the claimed benefits of XIML is that it can be used as a meta-meta language—a language that can not only be used to describe a user interface but also to describe all aspects of the design cycle, including the iterative process of developing a particular user interface design.

This wider view of the meaning of user interface design is demonstrated in a working example of a Meshoil program in section 5.1 on page 128. It shows how Meshoil can be used, not just to encode the final, working version of a user interface, but to support the various design steps in reaching this goal. The example is also a good demonstration of the natural progression from declarative to imperative programming.
4.3.4 XForms and XUL

XForms and XUL focus on creating GUIs for Web applications. In Meshoil, GUI creation is handled by using the API resources available on the platform where the Mesh engine is running. For an example of this, see appendix A.5.4 on page 312. This example also shows how GUI creation can be customised to suit particular situations—a feature listed as one of the advantages of XIML [34].

4.3.5 URC

UIDL’s are only proposed solutions to the universal usability problem. None have reached the stage of being fully implemented and widely accepted. (Universal Remote Console) [45] is the UIDL that currently comes closest to offering a practical, working system. In tacit recognition of this, URC was made an ANSI standard in 2005.

In section 6.1 on page 238, Meshoil goes head-to-head with the URC approach by taking URC’s own example, described in their ANSI standard, of a digital thermometer remotely controlled from a hand-held console. This working example demonstrates Meshoil’s greater flexibility as a user interface programming language, since the Meshoil program is able to:

- match URC’s description of console operation in a more abstract and generalised way—the declarative part of the system
- control the actual running of the thermometer—the imperative part of the system that lies beyond the scope of URC.