Metaphors we program by

Christopher Douce
Feedback Instruments, UK.
chrisd@fdbk.co.uk

Keywords: POP-V.A Metaphor,

Abstract

Due to the inherent abstract nature of certain types of software development, programmers and software engineers use metaphoric language throughout many areas of their work. This paper examines some of the many metaphors found within software development and engineering. A simplistic taxonomy is used to present the metaphors that have been found. The use and importance of metaphor and programming are discussed and some consideration is given towards the origins of metaphor. Intersections with other disciplines interested in this topic are also presented. It is concluded that software developers should ideally possess the ability to both understand and to generate new metaphors to successfully develop usable and successful software.

Introduction

Simplistically, a computer program can be considered to be a linguistic representation of a problem designed for two wildly different audiences. The first is the computer, the raw physical hardware that interprets instructions. The second is the human software developer or programmer. To the computer, the world is mathematical. For the software developer confronted with a world of numbers, useful support is required to aid in understanding what others may have written before. This paper explores the concept of metaphor in software, a topic that has been explored by other presenters at PPIG (Blackwell, 1996).

Lakoff and Johnson (1980) write, 'the essence of metaphor is understanding and experiencing one kind of thing in terms of another. Metaphor, it is argued, is prevalent throughout software development, and for a very good reason. Since software is something that is intrinsically abstract, software developers describe requirements and operations on data in terms of the world in which they are familiar. Developers, it is argued, use all their modalities to both write and comprehend software systems. Developers do not only touch the software systems they develop, they can, to a lesser degree, both hear and smell them. In doing so, they draw upon a rich vocabulary of representations which can then, in turn, be comprehended by others.

Metaphor in this paper is defined as 'a figure of speech or expression literally denoting one form of object is used in place of other to suggest similarity, likeness or analogy'. With this definition, we also consider words and phrases that are analogies, and in some cases similes. Just as some of the expressions used may be considered poetic, the author asks the reader for a degree of poetic licence.

The following section presents a brief survey of software metaphors that software developers should be familiar with. No real attempt to rigourously categorise the metaphors that has been made other than placing them is loosely related groups. Any number of sensible taxonomies can be constructed based on principles of similarity, usage or modality. Theorising about metaphor is also limited. This is left until the discussion section where one theory of the cognitive neuropsychological origins of metaphor is presented and then briefly explored. This is followed by a final conclusion that states that the understanding, appreciation and generation of metaphorical language within the area of software development and comprehension is a key, if not essential skill for the development of successful software systems.

2. Software development metaphors

Within moments of opening a text book on a programming language, you will be presented with a number of interesting metaphorical concepts. If the textbook is about Java, you will be presented with the ever present *object*. Moving onwards, you will be presented with an example of how to manipulate text. Text, you are told, is represented using *strings*. Imagine a set of paper cut-out letters, we may be told. If we have a needle and some thread, then we can push the needle through the centre of each cut-out character to form a *string*. Moving onwards, we find more beguiling ideas. Not long after considering the *string*, we are told to consider the *loop*. Is this really metaphorical? This is a loop that we cannot touch or feel. It is merely representative. Moving beyond our chosen programming language, we are gradually presented with a plethora of concepts, some of which are presented and roughly categorised in this section.

1 Traditional software metaphors

If we reach down, towards the *bare metal* of a computer, close to the machine code, we find assembly language. Here we find *flags*. We find *registers*. To look at our raw binary numbers we apparently can use *masks*. Flags indicate states, registers temporarily hold data and store results and interrupts indicate the need to respond to actions. It will not be long until you come across the *jump* instruction.

Moving a little higher, we find that we need store data in data structures so that we can get work done. We open another text book, this time on data structures and algorithms. Again, moments after opening the text, we find ideas expressed through metaphor. We see a *stack*, a *heap* and a *queue*. Strangely, since this is software, we find words that allude to physical actions. Data is *pushed* on to a top stack and something *pops* out of the top when we later want it back again. The *stack*, *heap*, *queue* and even *bucket* are all metaphorical containers (and all of them may potentially be prone to *leaks*, especially the *bucket*). Turning to the next page we find reference to *trees*. A *forest* is a data structure or store that contains many trees. Reading the section on trees had lead us to the part of the books regarding algorithms. Trees, as we all know, can be *pruned* and have *roots*.

Metaphoric language is used vigorously to describe algorithms. Sorting techniques is a particularly telling topic. Their names are deliciously evocative (flavour and programming will be mentioned later). We have the *bubblesort* and *shell* sort. Examining searching algorithms provides us the painful sounding binary *chop*.

Actions are not confined to pushing and popping, but are also extended to movement. *Walking* is a metaphor that is sometimes used with program debugging or tree traversal. Programmers use *heap walkers* to examine the state of variables. Dependency walkers are used to view function prototypes for dynamic link libraries. When processing goes astray, a system may halt or even end up in a *race condition*.

When reviewing another developers work, we may begin a structured *walk through*. On our journey, we may be presented with the most relevant and interesting landmarks, where we may ask relevant questions. If the going gets tough and we need to go slower, rather than walking, we may use a debugger to *step through*, into and out of particular lines of code. Finally, when we are sure about what we expect to happen, we may click on a traffic light icon and *run* our program.

The success of journeys from one place to another depends heavily on the weather. A system is said to be not functioning if it is *frozen*. The X.25 *cloud* has given way to an internet cloud in networking diagrams. A *broadcast storm* can arise on local networksegments. Shades of weather are do not appear to be particularly common, although a network administrator observing bandwidth usage may be tempted to explain that his network is *drizzling* or *spitting packets*. When organisation beyond the packet is required, packets are inserted into *frames* before being *delivered*.

A particular processor architecture has become commonly known as a *platform*. Alternatively, the term *environment* is commonly used, meaning a place where software can reside. Environments can be permissive, as well as restrictive (but not totalitarian).

The term environment is sometimes used in association with a *host*. A *benevolent host* is one which is generous to a process. Hosts can be kind, but only if you play by their rules. If something unexpected occurs, this is something exceptional. *Exceptions* must be *caught*. Programmers familiar with Java and C++ will be familiar with *trying* blocks of code. Processes can consume and have *forks* (food metaphors will follow). Threads are components of execution within a process. Other developers have taken this metaphor further, proposing an element of execution called a *fibre* can exist within a thread[1]. These terms may conjure up images of comfortable bed linen, allowing us to recall another useful metaphor - the concept of a *wrapper*. Software, like human relationships, sometimes work using hierarchy. Wrapping up software into different *levels* of abstraction is an approach that should be understood by all good engineers.

For those with exotic hardware, processes can have *affinity* with a particular CPUs. Such a term is considered to be apt, since affinity can be defined as 'having a harmonious relationship'. A simpler term may have be *association*. *Amity* could have been a similar candidate. Conversely (and interestingly), computing contains a degree of morbidity. Some metaphors are redolent of doom and gloom: administrators sometimes talk of *death* of processes. Processes not performing can be *killed* or *terminated*. If they cannot be killed, they become *zombies*, half alive, half dead. *Daemons* perform mystical operations such as printing and handling internet requests. Processes can also *panic*. *Collisions* can occur on both on virtual memory pages and on a network. A processor can be *hijacked* by a higher priority task. *Execution* and *hanging* in software, however, are not synonymous.

One of the most common metaphors in operating system use is the pervasive *wildcard*, the joker in the pack, allowing a command *prompt* user to specify which character should be ignored, or replaced as something different.

Zoological and biological metaphors

Zoological metaphors are present in software development. In colloquial language, dogs have a very hard life. Just as one can be "dog tired" at the end of a day, leading a "dog's life", trying to understand a "dog's dinner" of an installation, the resulting software that you patch together may "run like a dog". Dogs are also virtual. *Watchdogs* observe certain types of actions or processes to ensure that everything functions successfully, not unlike our tireless worker thread. If processes misbehave, the watchdog pounces, trying to remove a troublesome misnomer.

Spiders, of course, traverse world-wide-webs, collecting information. Design patterns make reference to *flies*, referring to their weight, rather than their ability to spread disease or to be incessantly annoying. Those who studied operating systems will remember the mythical *round robin*. There is, of course, the most famous zoological artefact in computing - the ubiquitous mouse. It is also possible to hypothesise that software developments that have been badly managed correctly or fail due to changing market conditions may become *white elephants*.

Within large installations, machines become animals within server *farms*, each machine performing a similar and requiring similar treatment. Having many different types of systems in your computer room may potentially be called a *server zoo*. Some environmental metaphors do not work. One term that has is considered to be silly is that of a *web garden* (heard at a conference held by a popular operating system vendor), evidently a mixed metaphor. This metaphor was used in a marketing context to distinguish one vendors solution to another vendors *web farm*. Farms are often dirty, smelly places, filled with animals and trouble. Farms require a significant amount of hard work to ensure that they remain serviceable. Gardens, by contrast, require less work. Following this line of reasoning, gardens may be considered to be a folly, a fertile patch of earth used primarily for pleasure. A garden may be aesthetically pleasing, but it may not yield very much. Farming, by contrast, is gardening on an industrial scale. Farming is an activity that can only be performed by professionals, whereas gardening is an activity that can be performed by amateurs.

[1] Any Prolog programmer *worth their salt* will know that the smallest element of a Prolog program is called an *atom*.

Parts of the body are also used: a *thumbnail* is a representation of a larger graphic. *Handshaking* occurs when we use our networking protocols. We are, of course, aware of the ever-present *window*. Some of us may be less aware that they can have *skins*.

Food, smell and taste

At some time or another, professional programmers will stumble across a portion of *spaghetti code* that inevitably drives them nuts. Food stuffs and computing are not closely associated since eating or drinking at the same time as using a PC is likely to cause human-computer interfacing difficulties. Coffee cups do figure, but we can surmise that this is due to a language designer potentially hoping to *kick start* its usage. Coffee *beans* considered as reusable components may, to some, be distasteful.

Food and actions surrounding food makes its appearance in other ways. Software developers can use *numerical recipes* to rustle up solutions to problems. In doing so, the authors may have to stir in an element of *syntactic sugar* to ensure that elements of their software *bind* well together. End users are continually presented with *menus*. Stressed system administrators are sometimes faced with the difficult problem of process *starvation*. Following continual development and addition of more items to menus, some software packages can, in contrast, become *bloated*. Data can be *harvested* and results even stored in a *warehouse*.

Operating systems can have *vanilla* installations, which require modification so that they can be *locked-down* against intruders. There is also little doubt that some software products leave both users and developers with a bitter taste in their mouths. When discussing refactoring Fowler (1999) refers to *bad smells*, elements of source code that are considered to be less than *sweet*. Developers are encouraged to excise malodorous fragments of code (and even data) whenever possible using a various array of simple techniques.

Machine or industrial metaphors

One much loved metaphor in computing is the *engine*. The word engine conveys a mechanism that provides power. An engine is also a mechanism that is potentially noisy, dirty and dangerous if not controlled or maintained properly or handled by appropriately trained individuals. Engines do not run without assistance. They require *tuning*.

In computing, the most common engines are *search engines* and *database engines*. Other forms of engine include *graphic engines* for three-dimensional rendering of landscapes, and *physics engines*, used to co-ordinate collisions in modern interactive games.

Historically, *wheel* has been used to refer to users with high privileges. For everything to function, for wheels to do work, to make sure that the engines are effectively used, we need some form of *driver*. Device drivers are commonplace within all modern operating systems, protecting the system from *crashes*. Like every good mechanic, a software engineer has a wide set of tools at his or her disposal. Rather than buying a sprocket set, a software engineer may build an adapter set. Repetitive data gathering and watching activities are performed by worker threads which may run forever, never tire, and very rarely go to sleep();

Compiling is also a metaphor, as is an *interpreter*. Optimisation can be *aggressive* (and software can, of course, be used in *anger*). All programmers will be familiar with the term *library*. Others will have heard of the notion of a data *dictionary*. *Page* and *book* metaphors are common. Virtual memory is divided into pages (an older technology was termed an *overlay*).

Within computer hardware, we have a number of interesting, simple and very comprehensible metaphors. A bus is a vehicle that shares a common route that takes many people to many different destinations. A bus is also a bunch of wires that shares a common route and takes many bits and bytes of data to many different destinations. Bytes have to wait, just as people do (when it is idling), so it can be used successfully. Continuing this exploration, we also apply real world attributes to a software (or hardware) *interrupt*, and the concept of a memory *bank*.

All these terms almost causes us to hear the loud sounds of industry and manufacturing; grinding, crashing and banging. Software, however, is mostly silent. Sound enters the programmers and administrators vocabulary when our computing machines are faced with increasing stress. Whilst servers may not roar with work whilst under pressure, we still like to attribute sound to them, saying that they are *creaking* and may even *crash*.

Software can be developed in a *clean room* environment, no doubt a crossover from integrated circuit manufacture. A number of papers have also discussed the development of *software factories*, although many practitioners would take issue with software being something that can be easily mass produced. Conversely, the metaphor of a software design *studio* is becoming increasingly popular. The word *studio* naturally inspires associations with creativity and innovation.

Service metaphors

Within network systems and on desktop systems daemons provide a plethora of services. Perhaps this is an example of language evolution or metaphor adoption and selection amongst computing practioners. The term *service* is very well used. It can refer to a printing service, or it can refer to a 'web service', an innovation that has only just been recently popularised. Services, as we know, are closely associated to servers.

Changes in technology has resulted in changes in vocabulary. We live in a world populated by thin clients, fat clients, smart client and rich clients (but, tellingly, not dumb clients or poor clients). Interesting, the world of the computing metaphor has been subject to forces of political correctness. In some circles, the term 'master/slave relationship' is considered to be an appropriate way to describe the operation of a software system, to the lay listener such expressions may suggest images that are less that palatable.

In many cases, the implementation and development of software systems is mediated through the establishment of formal contracts between the organisation that requires the software and the organisation that is to provide the services. Agreements are formed regarding cost, terms of delivery and sets of mutually acceptable terms and conditions are proposed. The idea of a *contract* has also found its way in the sphere of software design, specifically object-oriented programming.

Design by contract (DBC) is a programming technique which aims to increase software quality and reliability. In object-oriented programming a class and all its clients establish a 'contract' with each other. For a client to use a class the client must guarantee certain preconditions must be met before services are requested from that class. In the real world, the client must be an acceptable risk, having sufficient financial resources to satisfy the transaction. The reciprocal part of the contract being that the class will then guarantee that particular actions will be performed, namely, the contract will describe a set of postconditions. If both the preconditions and the postconditions can be represented in a form that can be checked easily by a language compiler, any potential violation of the contract, and therefore potential failings of a software system, errors can be detected, providing that the description of the contract is correct.

Whilst not strictly being a service metaphor, the use of DBC has some parallels with the notion of *assertions*, used in languages such as C, C++ and Java. With assertions, the programmer includes what can be described as a statement of fact within a program, essentially saying 'this must be true, before the following code is executed'. Assertions cause an error message to be raised, often in a rather dramatic fashion, drawing the programmer immediately to the problem in question. Usually, however, assertions are included during debugging, and are often removed automatically when production builds are created.

Development metaphors

Software development activities can be understood in terms of a number of different metaphors (see McConnell, 1993). We have the 'writing' metaphor, a single developer toiling over program prose. This metaphor is often not apt, since large software projects are team rather than individual efforts.

Again, we come across our *farming* metaphor. Software, it is said, can *grow*. There are limits to how far we can extend this idea before it becomes nonsensical. Crops need feeding and nurturing, land,

unlike memory space, should be left fallow to recover. One of the most suitable metaphor is the *building* metaphor. We now see software developers who are called software 'architects'. Architects build 'foundations' and use pre-existing, pre-fabricated *frameworks* on which to base their developments on (the .NET and J2EE frameworks are prime examples). The larger the project, the greater the number of specialisms there are: masons, carpenters, electricians, or DBAs, Java developers, graphical designers. Similarly, more care has to be taken to merge different components from other manufactures together.

Scientific developments are understood through the application of different models and metaphors to make sense of observed phenomena. As more is discovered, a different model may be found to be more apt. As McConnell writes, 'wrong' metaphors are not replaced by the 'right' metaphors, 'worse' metaphors are replaced by 'better' ones. As computing and software engineering develops we will continue to see different ideas being proposed, explored and examined, all through metaphor. Some ideas (such as relational databases) will be successful, other ideas (such as hierarchical databases) will be put aside.

User metaphors

The most obvious end-user metaphor is that of the desktop. In most cases, this metaphor is enormously successful. Documents are placed on top of the desktop, and moved to the wastepaper basket if no longer required. Desktop metaphors, if pushed too far become famously fragile. With early Macintosh systems, to have your disk returned to the user, the user must drag the disk to the same wastepaper basket – the same operation as to delete a page of text. A replacement with a the technical term 'eject' would have been equally bewildering to the novice user.

Not only do programmers work on their desktops with documents, sometimes they use workbenches. On other occasions, these may be called *workspaces*. To work successfully, they may need access to toolkits, accessible from *toolbars*. Recent innovations in HCI has produced the *wizard*, a faceless wonder which presents you with a series of dialog boxes. It is interesting to consider for a moment why the word 'wizard' was chosen in preference to the word 'witch'.

Some programming systems reply on a single comprehensible metaphor. Hypercard, a programming system used on the Macintosh platform uses at its heart a conception of a simple card, on which information can be filed and kept. Macromedia director uses an interesting theatric metaphor. *Events* (another popular programming abstraction) takes place on a *stage*, where the director can assign actions to individual components, which could be termed an *actor*. Directors have the ability to decide what happens, writers have to ability to write *scripts*.

Modern integrated development environments have taken inspiration from the arts. Instead of writing 'code', a programmer will paint a design, using a *palette* of reusable components. The metaphor is a useful albeit weak one, since a painter can mix together colours on a palette, but in modern IDEs a programmer cannot easily 'merge' components together to form a new component without resorting to a mechanism like inheritance. Every programmer will use the editors cut/copy/paste trio.

Domain specific languages such as LabView from National Instruments and Simulink from The Mathworks Inc. adopt a metaphor that has been derived from the environment in which they are applied. Rather than programming in the traditional sense of the word involving source code and compilers, parallels to electrical circuits are constructed, potentially making it easier for domain experts to adopt to the programming environment with relative ease.

Security metaphors

The longer one contemplates the different metaphors that are used within computing, the more examples one can find. Security metaphors are particularly enlightening since forms of security breaches are can be notably difficult to understand. In fact, the arena of computer security, a wondrous vocabulary has emerged.

Taking a leaf out of a physical security text book, IT security firms have developed *tripwire* software to facilitate intrusion detection. Developers *check in* source control in a similar way to how a traveller

may check in at an airport. Debuggers and security or error logs help to prevent software developers being lead up the garden path on a wild goose chase.

Most computer users will be aware of the potential of their systems being infected by a virus - a term that is an analog, rather than a metaphor, or by a more benign but potentially equally destructive *worm*. Systems are protected by *firewalls* (should these be 'fire breaks?'), to keep your own system safe from the dangerous mysterious world of the net, where rogue instructions and dangerous *trojans* are waiting to pounce if you ever let your guard down.

In an environment where viruses can replicate with rapidity and virtual *holes* can be exploited, the issue of control is one that appears firmly within our metaphoric vocabulary. As well as using watchdogs, we try to *police* our functions. When we wish to mark the end of a file, we use a *sentinel*. Parameters are personified when they need to be *marshalled* so they are converted from one form to another.

Exceptions can be raised if you attempt to break someone's *trust*. Trust can exist both at a machine level or at a code level. Code can be untrusted or trusted, and may be allowed to operate when holding an appropriate *certificate*. Code must behave. It should be thread safe. Within C++, functions can be *friendly*, but friends who do not behave in a structured manner could be considered harmful.

Systems administrators, some of them who adopt a *belt and braces* approach to system management, have a wide array of tools at their disposal. They can employ *sniffers* (not to be confused with watchdogs), build out *tripwires* and lure renegade crackers, *worms* and *viruses* into *honey pots*. In the game of defence, systems are *patched up* or *locked down* to prevent unwanted fiends from gaining access through old *back doors*. To exert constraints on running software, processes are sometimes allowed on to run in software *jails*, permitting only a limited number of actions. Certain software may be assigned appropriate software *policies*, allowing them certain *privileges*.

Metaphor directs our thinking, providing a frame of reference, priming us so that something can be understood. Security vulnerabilities may arise if elements of a system are used in a way that they were never designed to be used, something that hackers, crackers and phreakers excel at. "To avoid security vulnerabilities in your code, you must develop the habit of suspending, from time to time, your voluntary immersion in the program's metaphors" write Graff and van Wyk (2003). To find security holes, they reason, you must think like an alien and go beyond the developers original conception to find situations where your system may be abused. Since anything is possible within software, including changing the laws of physics, the use (and even overuse) of metaphor can be dangerous.

The metaphoric situation is becoming increasingly complex for our professional programmer. Recently, the term *firehose cursor* has been heard at a marketing presentation by the same operating system vendor that coined the expression *web garden*. You may get back your data from a web-service in a way like water is pumped through a firehose. The *cursor* in *firehose cursor* references a location within a *back end* system from where data will be squirted from. Danger can be attributed to fire. A firehose may be erroneously considered to be a security tool for a security firewall, for example.

Finding a new metaphor to describe a software development or administrative approach is desirable for a software vendor since it piques the interest of those seeking solutions to real-world problems. Marketeers must be careful. Instead of being a metaphoric crutch to aid comprehension, poor (or downright confusing) metaphors become badly designed sticks which have to be dragged around, serving no real purpose, slowing down progress.

Theories of metaphor

Metaphors come from our interactions with the world. Without metaphor, it would be impossible to talk about software. We use metaphoric language out of necessity, mapping our experience of the world onto our software designs. Our experience of the world is presented to us through our modalities: vision, hearing, touch,

smell/taste and proprioception. The software metaphors that have been described cover *all* these modalities. We *push*, *pull* and *yank* things. We may find certain function noxious, or sweet. Interestingly, we also have social metaphor, such as *jails* where we impose restrictions on others and *certificates*, as indicators of authority and truth, and *relationships*.

Since we appear to use metaphoric language so widely it may be informative to try and consider where metaphor comes from. One possibility, concerning synastesia, is proposed by Ramachandran and Hubbard (2001). Synastesia is a known condition where senses merge. Smells can be sensed as colours, sounds as tastes. One school of thought follows that our ability to generate and understand metaphors, particularly colourful ones, are due to a residual organic ability to understand different inputs from different sensory sources.

Consider simple metaphors for anger. People can become *heated*. Those angered may suddenly *explode*. Emotion are closely associated to the autonomic nervous system (ATS). When people become angered, it is perhaps no coincidence that these metaphors closely coincide with an increase in heart rate and blood-pressure. Those with depression *have the blues* a colour immediately associated with coldness. We can easily see similar merging of senses with software metaphor. One of the two most obvious cases being the *vanilla* operating system and fragments of source code being attributed with *bad smells*.

Just as we associate metaphor with modality, it is not unreasonable that metaphor can be associated with the self (see Watt, 1998). This can be found when we consider the role and action of variables. Comprehension of software and variable role is a subject that has recently received attention from other PPIG members (Sajaniemi, 2002). Put simply, variables can have a role of a *collector*, for example, creating a total for a list of numbers that are generated from an *iterator* or loop variable. Combining roles with identification is, no doubt, an incredibly powerful didactic aid.

A learner relies on existing knowledge and attributes of concepts that are rigorously understood. Understanding metaphor relies on similarity between the concept that is being discussed and the framework of an existing idea to establish comprehension. When contemplating similarly, we step into the arena of category, an area that is discussed by Lakoff and has been studied in great depth by Rosch et. al. (1975). When proposing a metaphor to assist in comprehension, the level of the representation used is important. Selecting a category or concept that it too specific (or high) may confuse, especially if some of the attributes are not particularly relevant to the idea that is being presented. Metaphors must be immediately accessible, and where possible, prototypical.

Discussion

As mentioned at the beginning of this paper, one of the first ever computing metaphor that we learn is the *loop*. This may be preceeded or followed by the notion of a *string*. If faced with such terms, it may be easier to comprehend software as something that is *woven*, rather then *written*. Indeed, we have seen that there are a number of different ways to perceive software development. Lakoff and Johnson propose that metaphors are not merely linguistic ornaments but are instead closely related to the structure of thought and cognition.

As computing power has developed, the more widespread object-oriented languages have become. Rather than develop a metaphoric representation to describe requirements and developing software in the level of the target hardware, object-orientation allows the software developer to code using the metaphors and concepts proposed during earlier stages of a software development life cycle. Metaphor is central to the naming of classes, objects, functions, member attributes and variables.

Metaphors are, of course, abstractions. A large number of useful programming abstractions can be found within the area of design patterns (Gamma et. al., 1995). Software developers sometimes talk of constructing *bridges* and building *façades*. Objects can be decorated. Objects can be created using factories, and *singletons* can exist on their own. Patterns can can describe groups of common relationships between objects (model, view, controller), and also describe how groups of objects are created (creational patterns) and work together. These abstractions are, of course, metaphorical.

Previous PPIG presentations (PPIG '99) have compared the concept of design patterns to the older notion of a programming plan (Rist, 1986).

Metaphor and the efficacy of program comprehension are intrinsically related. During the understanding of computer programs we can construct a comprehensible metaphor from what amounts to an abstract mechanical representation of a machine in the form of a computer program. Conversely, writing software requires a developer to *paint* a representation using constructs that are colours of our chosen computation language.

Whilst performing this somewhat introspective survey of the metaphors used within software, it was interesting to discover that the metaphors cover each of our modalities. Most obviously, programming requires use of our visual and spatial abilities, to comprehend diagrammatic notations and to associate sections of source code together. Similarly, auditory and linguistic abilities are drawn upon during the reading of identifiers and the generation of linguistic labels for software artefacts.

The use of multi-modalities within programming has been hinted at by Petre (2003) who explores the use of mental imagery within groups of programmers. Rather than using the regular 'what are you thinking?' question so often used within protocol studies, Petre instead used more interesting probes that could be considered to be modality specific.

One striking question that can be considered is whether the appreciation of computing metaphor is something that could be taught. It is obvious that we teach using metaphor. In computer science education, the selection of appropriate metaphor may make a difference between a blank face, and a comprehend concept. Beyond the classroom, towards the industrial arena, software development guidebooks actively encourage the use of an appropriate metaphor, emphasising that it should be disposed if it ceases to be useful (Beck, 2000).

Finally, it is sad that so many of the colourful phrases and descriptions used in software are so often anglocentric in origin, often of the American variety. Take the example *garbage collector*. In British English, the garbage collector would be termed either the *bin man* (or bin 'person'), or *rubbish collector*[2]. The vocabulary of expressions used to explain programming abstractions may become richer as the need for software and its development becomes more evident within other cultures.

Conclusions

Software development is rich with metaphor. Since software development is such an abstract activity the use of metaphor is considered a necessity. Some software design and development approaches embrace the idea that a software project can be guided by a central metaphor. 'By asking for a metaphor we are likely to get an architecture that is easy to communicate and elaborate', (Beck, 2000). Metaphor within computing extends beyond architectural considerations and permeate into almost every aspect of design, development and administration.

This paper is primarily an exploratory discourse, describing a number of common software metaphors that are easily accessible. A more systematic approach to software metaphor analysis is, of course, possible. The origins and application of metaphor is a fascinating topic and one that is of interest to those working within other disciplines including cognitive psychology (see Tourangeau & Sternberg, 1982), and linguistics,notably by Lakoff (1987), and Lakoff and Johnson (1980). Cross-discipline interest in metaphor has obviously arisen in the area of language translation from the field of computational linguistics. Some researchers have utilised the data processing capacity of computers to analyse selection portions of text, known as corpora (see Martin, 1994) to try to provide empirical evidence to support theories. Philosophers also have an interest in understanding what metaphor is, and how it relates to human thought and cognition (see Ortony, 1984).

Lakoff and Johnson argue that human cognitive processes are largely metaphorical. Software developers not only need to be mathematicians, logicians, engineers and scientists. They also need to

[2]The author of this paper has always played in *sand pits* rather than *sand boxes*, and would prefer his software to languish in a *gaol* rather than a *jail*.

be playwrites and poets. Software developers need to nurture their metaphoric faculties to express the abstract software forms using expressive language for greatly different audiences, both computational and human. As computing and information technology develops, so will its associated vocabulary of useful metaphors.

References

Beck, K. (2000) Extreme Programming Explained: Embrace Change. Addison Wesley.

Blackwell, A. F. (1996) Metaphor or analogy: How should we see programming abstractions? *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group*, 105-113.

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts, Addison-Wesley.

Graff, M. G and van Wyk, K. R. (2003) *Secure Coding : Principles and Practices*. O'Reilly and Associates.

Lakoff, G. (1987) *Women, Fire and Dangerous Things*. University of Chicago Press.

Lakoff, G. and Johnson, M. (1980) *Metaphors We Live By*. University of Chicago Press.

Lawler, J. (1999) *Metaphors we compute by. Figures of thought for college writers*. Mayfield Publishing.

Martin, J. H. (1994) A corpus-based analysis of context effects on metaphor comprehension. Technical report, University of Colorado. CU-CS-738-94.

McConnell, S. (1993) *Code Complete: a Practical Handbook of Software Construction*. Microsoft Press.

Ortony, S. (1984) *Metaphor and Thought*. Cambridge University Press.

Petre, M. (2003) Team coordination through externalised mental imagery. *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group*.

Ramachandran, V.S. and Hubbard, E.M. (2001) Synaesthesia - a window into perception, thought and language. *Journal of Consciousness Studies*,Vol 8, No. 12.

Rist, R. S. (1986). Plans in programming: definition, demonstration and development. *Empirical Studies of Programmers*. Ablex.

Rosch, E. (1975) Cognitive references of semantic categories. *Journal of Experimental Psychology : General*. 104, 192-233.

Tourangeau, R. & Sternberg, R. J. (1982) Understanding and appreciating metaphors. *Cognition*, 11(3), 203-244.

Sajaniemi, J. (2002) Visualizing roles of variables to novice programmers. *Psychology of Programming Interest Group*. 18-21 June 2002, Brunel University, London, UK.

Watt, S. (1998) Syntonicity and the psychology of programming. *Psychology of Programming Interest Group*. The Open University, Milton Keynes, UK.