EE3RCS - Individual Research Project

On dealing with complexity in modern application development

Chris Cummins

December 2012

Abstract

This study investigates various methodologies for dealing with complexity in modern application development. With a focus on open source technologies and language design, it provides quantified measurements of the effects that using higher level language constructs can have on software size and execution efficiency.

Contents

List of Figures

List of Tables

1 INTRODUCTION

1.1 Context

Intel Corporation is a massive multinational semiconductor manufacturer, with a distinguished list of achievements over the course of its 44 year history, from the shared invention of the microprocessor, the x86 instruction set, development of 14 nanometer scale transistor technology, popularisation of ingredient based branding and continued and aggressive advances in communications and computing technology for over four decades.

Based in Santa Clara but with offices distributed globally from Buenos Aires to Beijing, it is the largest and highest valued microprocessor manufacturer in the world. With 2012 Q3 revenue of $13.5 billion ($54.0 billion in the 2011 fiscal year) and a workforce of over 100,000 employees [18], the corporation maintains a dominant 16.5% share of the semiconductor industry. Of this, a 70% revenue majority comes from its PC business [11], with the company owning a 81.3% share of the notebook processor market.

However, with only a 0.2% share in mobile technologies [10], the company is facing increasing challenges from competitors such as Qualcomm with its 48% revenue share of the mobile market [29]. This is reflected in the market trend towards increasingly mobile consumer devices, a field in which design company ARM Holdings has secured solid ground supplying microprocessors for Apple iPhones, Windows tablets and the majority of Google Android handset manufacturers.

Despite the apparently head-to-head nature of Intel and ARM's businesses, it is important to note the distinctions between the two companies. While Intel is a manufacturer, ARM is solely a design company, leasing licenses for its intellectual property (IP) to third party manufacturers on their behalf (of which Intel is one). A more viable and direct competitor to Intel is Samsung Electronics of the Samsung Group, another manufacturer of ARM licensed designs. Over the same 10 year period between 1999-2009, Samsung's semiconductor sales rose 13.5%, compared to Intel's 3.4%, making it the second largest globally, and raising speculation over Intel's ability to maintain its position as market leader [9]; since Samsung has achieved leading position in production of mobile phones, LCDs, memory chips and Televisions.

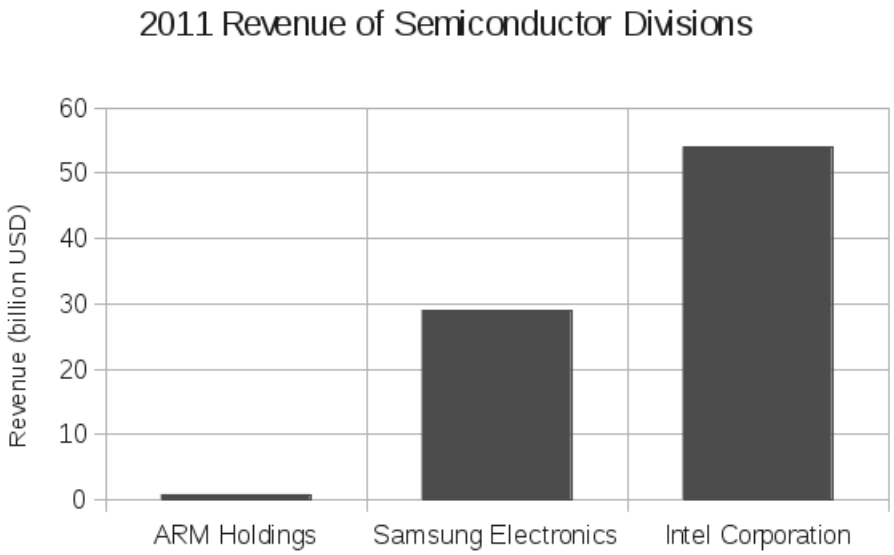

Figure 1: A comparison of 2011 annual revenues for Intel Corporation [18], ARM Holdings [3], and Samsung Electronics (semiconductor division) [32].

1.1.1 Technological Innovations

Intel has long been at the forefront of semiconductor innovation, with a reputation for investing heavily in research & development (R&D) to stimulate growth in emerging markets and fields of technology. In 1965, Intel co-founder Gordon Moore predicted that "transistor density on integrated circuits doubles

about every two years" [26], a statement which became popularised in mass culture as 'Moore's Law', being generalised to meaning that computers will get twice as powerful every two years. An examination of the available data shows this prediction to be largely accurate, as shown in Figure 2.

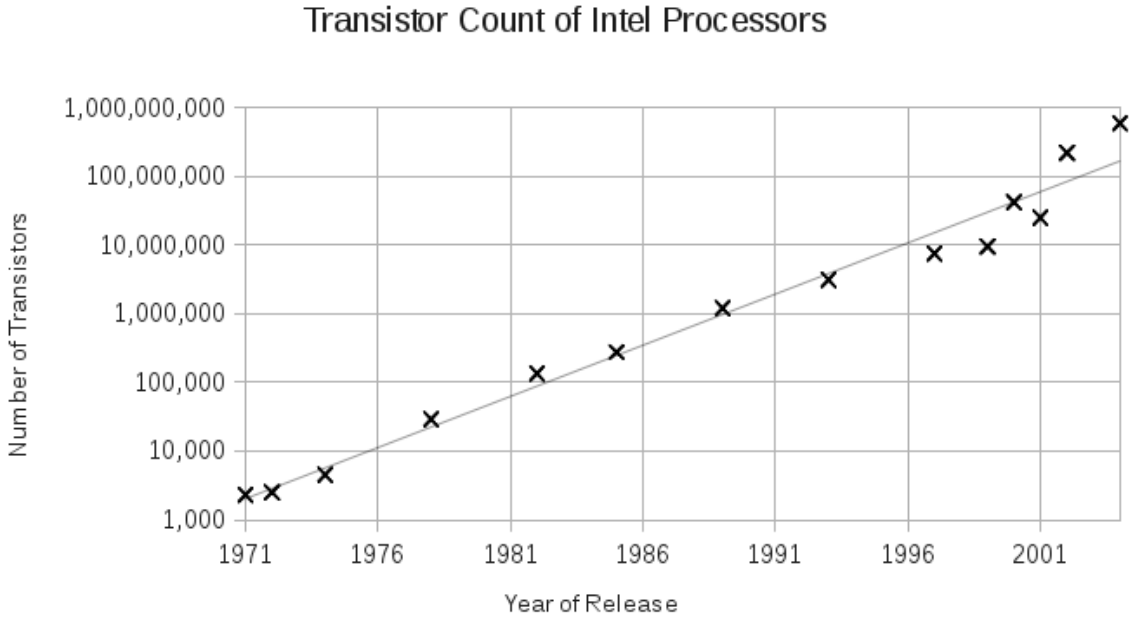

Figure 2: Number of transistors in Intel microprocessors over time (see Table 1 for data). Note the logarithmic vertical axis, showing the exponential increase in transistor density.

1.1.2 Open Source Technology Center

A contributing factor to Intel's success is the large investments made into providing a high quality of support for its products. By investing heavily in software, it can offer a strong level of usability for its hardware products, leveraging this advantage to gain customer satisfaction and to build a core sense of brand loyalty.

One of the areas of heavy investment is in Intel's Open Source Technology Center (OTC), a department of the Software and Services Group (SSG). Of this, particular focus is placed on support for the Linux kernel and GNU/Linux operating systems. Since its creation in 1991 as a personal project of then University of Helsinki student Linus Torvalds, the Linux kernel has grown to become the most popular of the UNIX-like kernels, with a 50.6% share of UNIX kernels for web servers [37], a field in which 64.6% of operating systems are UNIX based [38]. Intel have been heavy investors in Linux since 1993, after starting an "Enterprise Grade Linux" initiative that secured its position as being among the top 10 Kernel contributors, as well as contributing to many other open source solutions such as the Yocto Project, a widespread suite of tools used for creating Linux Distributions on embedded systems.

The influence that the OTC has had on promoting open source software on Intel hardware has helped secure its position as the largest manufacturer of server technologies in the world. This has the mutually beneficial effect of improving the quality of readily available free software while boosting Intel revenue and brand loyalty.

Directed by Imad Sousou, the OTC consists of over 400 employees, with the main offices located in Portland, Oregon. Within this department, the office at St Clares, London presents the main UK base of operations, with a larger corporate office at Swindon being one of the headquarters of Intel's EMEA (Europe, Middle East, Africa) sales region.

Within the London office, there is an approximately even split of personnel between designers and engineers, with the designers comprising of visual and interaction specialists, and the engineers being distributed among a number of teams such as graphics, visualisation and kernel developers.

While the majority of open source development at Intel is focused on providing hardware support for Intel components, there is also a focus on developing end user software. Recent projects include

the MeeGo and Moblin operating systems, which are open source GNU/Linux based operating systems targeted at mobile devices. This software was used to leverage the popularity of lower powered netbook devices powered by Intel Atom processors by offering a stable and optimized operating system. The MeeGo operating system was a joint partnership between Intel and Nokia, combining efforts from their respective Moblin and Maemo platforms, creating a Debian/Fedora based Linux distribution with a Clutter/Mutter user interface [21].

After the project was discontinued in mid 2011, the successor to the MeeGo operating system is Tizen, a standards-based software platform being developed by Intel and Samsung and aimed at mobile devices such as smartphones, tablets and In-Vehicle Infotainment (IVI) systems.

1.1.3 Aldgate Prototype

The London OTC office began work on a prototype user interface (UI) for the Tizen handset operating system in July 2012, code-named Aldgate. The design team created wire-frames, mock-ups and assets to demonstrate the design, while the engineering team began implementing this design as a prototype UI for use on test handsets, creating a feedback loop between the designers and engineers wherein the effects of design decisions could be tested in real world scenarios on real phones.

The other intern and I joined the prototype development team at the end of August 2012 and began working with the other three engineers on implementing the prototype as a native Android application, working with the designers to ensure it closely matches their intended design and reflects their decisions.

The prototype represents a large end user application - a full interface for a mobile phone operating system, responsible for managing users' contacts, address books, applications and handset activity. Given the small team size and tight project deadline of approximately 6 months, the success of such a demanding project is dependent largely on the ability to minimise and manage complexity in software.

1.2 Aims

It is a demonstrable truth that software projects have fallen as a result of poor complexity management, and it is not just small or low budget efforts that are susceptible. Perhaps the most prominent example of this is IBM's OS/360, a project with such disastrous consequences in terms of budget, customer satisfaction and organisation that it is arguably responsible for the birth of software engineering as a discipline in itself (this is the subject of Brook's *Mythical Man Month* [5]).

The aim of this study is to provide an evaluation of the problems facing developers in managing software complexity. By focusing on the challenges experienced by developers of modern application software, a number of problems and widespread solutions with be examined, with the intention of providing insight into ways of simplifying the creation of modern applications.

1.3 Objectives

In order to archive the stated aims, the study will be guided by three objectives:

1. To discover the driving motivation behind increased complexity in software, and to analyse the quality and success of existing approaches to complexity.

2. To investigate the consequences that these approaches to complexity have on software, performance, and programmer productivity.

3. To formulate an expandable approach to managing complexity in software application projects.

1.4 Hypothesis

It is my contention that complexity is not a necessity of software, and that the illusion of complexity which surrounds software engineering is the result of rapid growth in a young discipline, and is not an integral feature of the subject.

1.5 Scope

The scope of this study limits itself to modern application development. While this is arguably a broad umbrella topic, two further restrictions are placed upon the scope: it is limited to software products which interact directly with the user, and products which are in active development today. By imposing these artificial limitations, it is possible to exclude administrative software such as web servers and system daemons as they do not interact with the average user and so have differing design goals. Furthermore, this excludes anything without a user interface such as kernel space software.

1.6 Overview

After a brief review of some of the historical related literature, the body of the study is divided into three sections, each addressing one of the objectives (see page 3). Furthermore there are four case studies providing supporting investigations and data.

1.7 Conventions Adopted

The Vancouver system will be used throughout to reference other texts. Page 29 contains the list of references.

Acronyms will be used where appropriate in place of full terms. When one is to be used, the first use of the term will include the full expanded name, followed by the acronym in parenthesis. The acronym will then be used from thereon in.

The term "UNIX" is a legal trademark of The Open Group, however, it has been adopted in various forms (Unix, *nix, UN*X) to refer to the full family of UNIX-like systems. This term covers any operating system which behaves in a manner similar to that of a UNIX system, while not necessarily being recognised or licensed by the The Open Group. Throughout this study, the terms will be used interchangeable to refer to the full family of UNIX, GNU/Linux, BSD, MINIX and other systems. Additionally, the term "Linux" is used to refer to the GNU/Linux operating system, often as a fully fledged distribution complete with graphical interface and end user applications.

2 LITERATURE REVIEW

Given the importance of the topic, it is of no surprise that much has been written on the topic of complexity, and it has been the subject of much research. Of these works, some of the most recognized - Brooks' *The Mythical Man-Month*, Fowler's *Refactoring* and Martin's *Clean Code* are works soley dedicated to the topic, though there are a number briefer papers and other publications with sections on the topic.

An examination of some of the more popular publications reveals fundamental shifts in the exact meaning of the term 'complexity' over time. Evidence of early attempts to manage software complexity can be found in McCabe's 1976 paper [24], in which he argues against phsical program size as a metric of code complexity, using a simple - if rather contrived - example of 25 sequential flow switches in a single program. By chaining flow control in such a way, the resulting program could have as many as 2^{25} (over 33 million) different code paths, an almost impossible program to achieve 100% test coverage of. Furthermore, this physical size restriction on programs would encourage larger and more complex processor instruction sets, as processor architects create ever more complex instructions in place of a few simple operations. At the University of California in 1980, Dr David Patterson proposed the case for the Reduced Instruction Set Computer (RISC) based on the argument of complexity management.

"With early computers, memory was very expensive. It was therefore cost effective to have very compact programs. Complex instruction sets are often heralded for their 'supposed' code compaction." [28]

These examples indicate a historical desire for complexity, one born of necessity and resource management, an irrelevant demand now, as memory is both fiscally and computationally cheap. This desire for performance at the expense of increased complexity is not uncommon, as demonstrated by the rise and fall of the `goto` statement, a function too crude to be worth the cost in code obfuscation that entailed from its use. As Dijkstra asserted, "The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program" [8].

It was later identified that one of the leading tools in complexity management is abstraction, and authors such as Fowler and Miller garnered much respect for publishing works on managing this abstraction, with Fowler's *Refactoring* being the first dedicated teatrise to the art of pure code complexity management, identifying the need for code to be primarily human readable.

"The compiler doesn't care whether the code is ugly or clean. But when we change the systen, there is a human involved, and humans do care. A poorly designed system is hard to change. Hard because it is hard to figure out where the changes are needed. If it is hard to figure out what to change, there is a strong chance that the programmer will make a mistake and introduce bugs." [12]

Perhaps the most significant and proactive steps in managing software complexity have come about as a result of the Agile software methodology, which came about as a response to the traditional waterfall style software development processes that Miller believed to be stagnant and inefficient. From the movement, the Agile Manifesto (Figure 3) has become something of a mantra for modern software development.

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

Figure 3: The Agile Manifesto [23].

Despite this, there are shortcomings in the Agile movement in that it does not offer any direct or explicit methods for managing complexity, but rather offers a development process which intends to reduce the risk of incurring unecessary complexity to begin with.

3 ABSTRACTION

3.1 On Complexity

"Controlling complexity is the essence of computer programming." [19]

Traditionally, computer science has been a rather hostile and elitist area of academia with high barriers to entry from scarcity, large space requirements, high prices and vast complexity. Over the last three decades, tremendous effort has been placed on increasing the accessibility of computing, and with the aid of initiatives such as the BBC Micro and educational reforms, general computer literacy has increased exponentially. Reductions in complexity can be largely attributed to the advent of serious and conscientious development in user interfaces which act as buffers between the user and the complexity of the underlying hardware. Of particular note is the advent and refining of the Graphic User Interface (GUI), as summarised by Togazzini:

> "On the original electronic computers, there was no metaphor between the user and the raw reality of vacuum tubes and wires. In today's visible interfaces, that raw reality has been replaced with a softer, virtual reality - an illusion spun of nothing more than light and logic." [35]

This illusion presents itself in the form of common metaphors that have ingrained themselves in modern technology, from the mouse and keyboard as input devices through to common abstractions such as the desktop, icons and files. Advances in the field of Human-Computer Interaction (HCI) are responsible for the modern emphasis on software that is designed for recognition rather than recall. Users can now navigate and interact with devices on which they have no formal training, a far cry from teletype terminals and batch processing operating systems, which could not be operated without appropriate training.

3.1.1 The Cost of Ease of Use

This decrease in end user complexity was not achieved without cost. Rather, there was an offloading of complexity from the end user to programmer. Instead of relying on the user to have intimate knowledge of the hardware and task being attempted, software began being responsible for automating more tasks. Devices became more 'plug and play', applications became more interactive, and the whole end user experience became more intuitive. In essence, software became smarter so that users can be dumber.

To demonstrate this, consider the logic flow of a simple application to query a dictionary. In the first scenario (Figure 4), the application is a non-interactive batch application. It has a linear progression from start state to end state with two branches for error conditions.

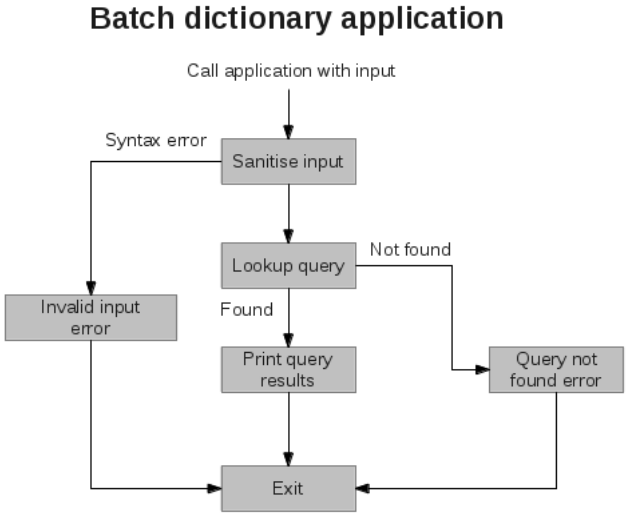

Figure 4: The available code paths in a batch dictionary application.

```
$ dict lookup duck
  A waterbird with a broad blunt bill, short legs, webbed feet,
  and a waddling gait.
$ dict lookup dog
  A domesticated carnivorous mammal (Canis familiaris) that typically
  has a long snout and acute sense of smell.
```

Figure 5: An example session with a batch dictionary application.

By comparison, consider an interactive dictionary application look-up (Figure 6)

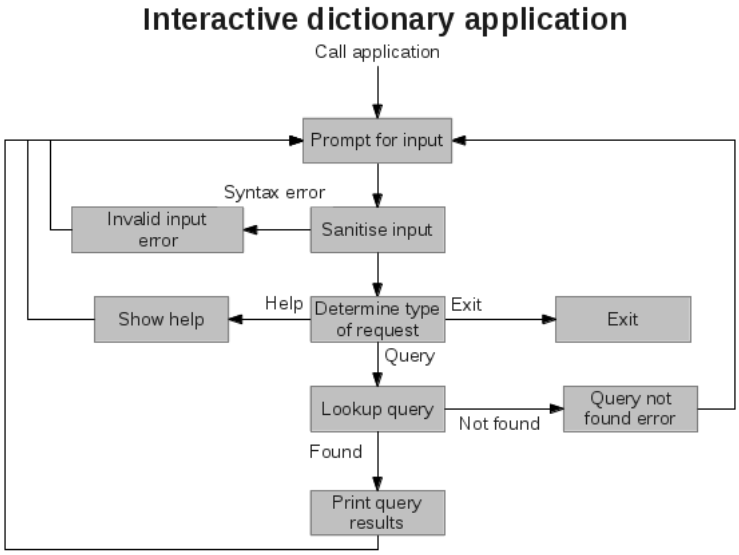

Figure 6: The available code paths in an interactive dictionary application.

```
$ dict
  Command <lookup | quit | help>: > lookup
  Lookup word: > duck
  A waterbird with a broad blunt bill, short legs, webbed feet,
  and a waddling gait.
  Command <lookup | quit | help>: > lookup
  Lookup word: > dog
  A domesticated carnivorous mammal (Canis familiaris) that typically
  has a long snout and acute sense of smell.
  Command <lookup | quit | help>: > quit
```

Figure 7: An example session with an interactive dictionary application.

Graphing the paths through a single iteration of the look-up query (Figure 8) reveals the extra branches in the interactive version, and calculating the cyclomatic complexity[1] for both reveals that this extra interactivity for the users benefit comes at the cost of a 125% increase in cyclomatic complexity, leading to more code and so a greater opportunity for bugs.

$$M = EN + 2P$$

$$M_b = 4 - 6 + 2 \times 3$$

$$M_b = 4$$

[1] Formula derived in McCabe's 'A Complexity Measure' [24]

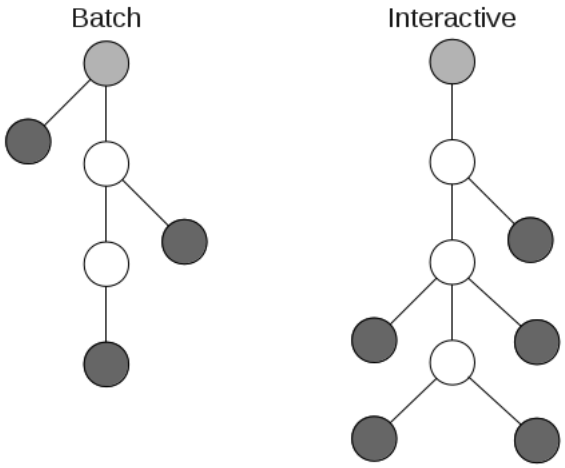

Figure 8: A comparison of cyclomatic complexities for a single iteration of the batch and interactive dictionary applications.

$$M_i = 8 - 9 + 2 \times 5$$

$$M_i = 9$$

Of course this example is over-simplified and limited in scope. In reality, the complexities of modern GUIs in applications can present the majority of development effort and time. Much work is done to ensure accessibility across different user demographics, so as to prevent the trap of making too many assumptions about a user-base; "Design with only a single user in mind, and you will find that only a single user can use your program." [35] This means that interface logic can be responsible for the majority of complexity in an application.

3.1.2 Defining Complexity

Complexity is an often used term in software engineering so it is necessary to formalise a definition before proceeding into a discussion of it. In a non technical context, complexity is tightly coupled with quantity. A complex system is one with many intricate parts, and a complex calculation is one which requires many lesser steps. Conversely, 'simplicity' is that is easily understood and intuitive. Applying this definition to software, it is clear that reducing the number of apparent 'components' in a piece of software is instrumental in complexity management.

As Miller showed, the number of objects that an average human can hold in working memory is 7 ± 2 [25]. By finding a suitable metaphor for 'objects' in software, a quantified measurement of software complexity can be made, with the target value being ≤ 7. By achieving this goal, it can be reasonably assumed that a programmer may maintain every one of these objects in their working memory at any time, and so have a comprehensible piece of software. This means that complexity in software can be defined as a piece of software that a single programmer could not reasonably be expected to understand, as it consists of too many intricacies and parts.

3.2 On Computable Ideas

"Before there were computers, there were algorithms." [6]

Turing's milestone publication[2] - to which this chapter owes its title - has survived as a prophetic work on computation for almost 80 years, an unusually long lifespan for a piece of technical literature. This can be traced back its sweeping use of abstractions, still offering a sound insight into the science of computing with the conceptual 'Turing' machine today, a field in which most works are quickly relegated to the realm of scientific novelty (consider the primitively crude ENIAC or the naive curiosity of Babbage's difference engines).

[2]On Computable Numbers, with an Application to the Entscheidungsproblem [36]

This is a recurring pattern in the history of computing; while many of the field's greatest achievements seem simple and occasionally even quaint in light of today's technology, certain works achieve that truly coveted goal of being 'timeless'. Further investigation reveals the distinction to be one of product vs processes. While the aging of electronic products is rapid and ungraceful, the processes through which they are programmed only compound.

That is not to say that software does not date (as even the most half hearted review of software history will surely reveal), but that the process of software creation does not age, a property that is the result of abstraction, and one that Brooks romantisises - *"The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination"* [5].

As a result, it could be viewed that in many ways the terms software engineering and computer science are a bastardisation of the language, since software engineering is a formal science dealing with abstract logic and mathematics, and computer science is a subset of electronic engineering, forwarding the advancement of (and being limited by) the technology of today.

3.2.1 Abstraction in Software

A prominent example of a long lasting piece of software is the UNIX operating system (OS), developed over 43 years ago by a small group of engineers at AT&T's Bell Laboratories. While the original PDP-7 assembly implementation is long since obsolete, the innovations of the original project - chiefly, the design of a full application programming interface (API) for the OS - have ensured its longevity, by abstracting the hardware from the software that it runs. Over the past four decades, this has expanded and standardised to the point where "Unix is still the only operating system that can present a consistent, documented [API] across a heterogeneous mix of computers, vendors, and special-purpose hardware" [31].

It is arguable that the longevity of a piece of software is directly proportional to it's use abstraction; a point that is emphasised strongly by the quality of UNIX's API, meaning that "Binary-only applications for other operating systems die with their birth environments, but Unix sources are forever" [31]. An additional advantage of the proper use of abstraction is that of software portability. By abstracting the differences in hardware from the OS API, a truly hardware-independent standard for portable software results, meaning that the same software which runs on supercomputers [15] can also be used to power educational devices such as the Raspberry Pi, through to some the world's largest server farms [2]. As Raymond notes, "The Unix API is the closest thing to a hardware-independent standard for writing truly portable software that exists" [31].

This key abstraction provided the required longevity give UNIX real market traction, with an estimated fifty million man-years of development put into it [31]. The result is that it expanded in many ways far beyond its original specification. For example, the Linux kernel, the most popular open source Unix kernel has grown from 14,000 lines of code written by a single author through to over 10 million lines of code[3] spread across a global authorship of at least 500 developers and dozens of corporations. This means that it can not be reasonably expected for any one person to have an intimate knowledge of the whole project (which in itself is just a minor subset of the functionality of a modern operating system).

Coupled with Moore's law and other technological advances; the unfortunate but inescapable realisation is that modern software engineers are developing on hardware they don't understand, working on projects they can't comprehend.

3.3 Everything is an object

As with most disciplines, software engineering can be seen to have a status quo which goes through phases and trends. It is certainly evident that the most recent innovations in software have been bringing about the popularity of object orientated programming. Where there were procedural and

[3]Linux kernel source count derived from 3.x release, available `http://www.kernel.org/`. Actual physical lines of code: 10,519,748.

functional programming languages, the commonly accepted status quo now is focused upon object orientated languages. Object orientated programming shows an attempt to deal with software complexity by modularising everything to point where 'everything is an object', and then giving these objects responsibility over their own behaviour, essentially giving each object its own domain. This modularisation technique has become hugely popular as a means for splitting up large tasks and managing projects, with Figure 9 showing that 70% of the most popular language choices for public GitHub repositories are object orientated languages.

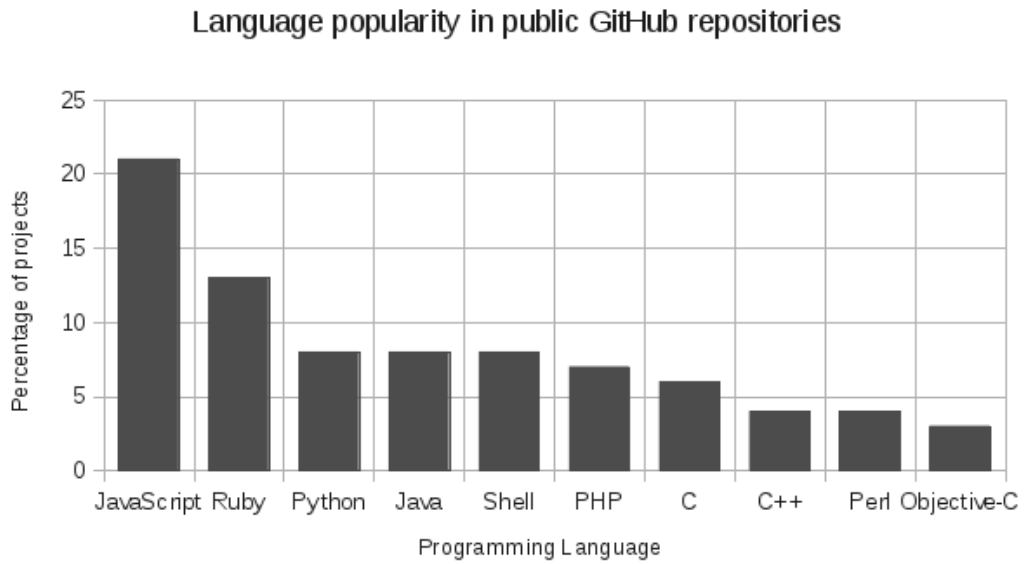

Figure 9: The 10 most popular programming languages of public GitHub repositories, see page 24 for a full table of results.

Interestingly, by plotting the same data-set (minus the 'Shell' option as that is too ambiguous to be useful, not referring to any single language) chronologically in order of the language creation, a new trend appears, with almost a perfect correlation between year of language creation and it's popularity (Figure 10).

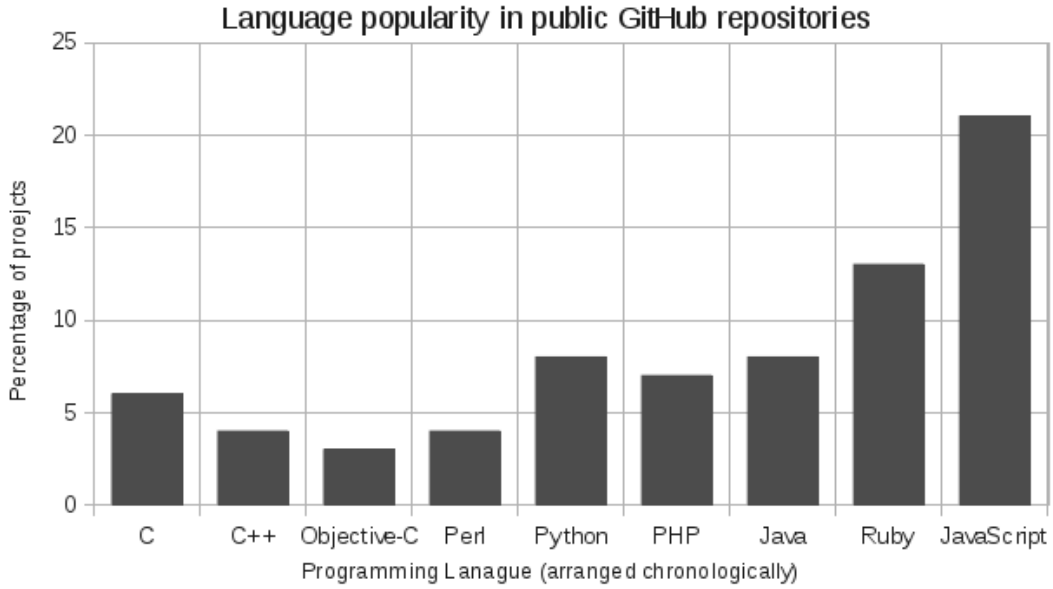

Figure 10: The 9 most popular programming languages of public GitHub repositories, ordered chronologically by year of language creation.

There are multiple interpretations for the cause of this pattern, one of which is that it is indicative of a refining set of requirements from application developers. What passed as an acceptable language specification many years ago may no longer be up to par. This idea is reinforced also by the expanding

size of languages themselves. JavaScript is a much more expansive language than C, which is Spartan by comparison. Another interpretation for this correlation is that the modern languages are achieving something that was not technically possible with the earlier languages, or that this extra performance allows for a certain degree of freedom about the level of optimisation that is required to achieve satisfactory performance. It is definitely true that the machines have become more powerful by orders of magnitude[4], and this could further be reflected in the increase of language specification size.

3.4 Case Study A

> *"In a time when the latest fads in programming tend to be object-oriented high-level languages implemented using byte-code interpreters, the trend is clearly to learn to write portable programs with high reliability in record time. It seems that worrying about memory usage and CPU cycles is a relic from a by-gone era."* [33]

Seyfarth offers a cynical interpretation of the drift towards higher level languages, but it is an opinion that can be backed by observable truths. While working on the Aldgate prototype (a native C application), the decision was made to re-implement the entire codebase in Vala - a new programming language developed by the GNOME Foundation with the specific intention of offering modern programming language features to GNOME developers.

Vala is a high level language with a C# inspired syntax, providing namespacing, classes, objects and deterministic memory management through automatic reference counting. The result is that the programmer has the ability to write C code in a higher level environment.

The rewrite from C to Vala involved re-implementing a full suite of features, objects and classes, requiring an estimated man-month's worth of development effort. Given the tight schedule that the prototype project was operating under, it became clear that there must be significant advantages to Vala that made this decision worthwhile.

An examination of the Vala language shows it to be a good candidate for an AB comparison with a lower level language. The Vala compiler takes an input Vala source file and from this generates C source. This means that by comparing the size of the Vala and Generated C source files, observations can be made about the efficiency of the two languages.

Figure 11 shows a comparison of the non-whitespace character count of each of the Vala source files in the Aldgate prototype, compared against the size of the generated C file (displayed as a multiple of the input size). See page 25 for a full table of results and methodology.

Some interesting conclusions can be drawn from the chart. First, the size of the C source files tends towards around 500% of the input Vala source file. Secondly, there is a sharp increase in proportional C size as the Vala sources drop below 500 chars. This second conclusion is of particular interest, as it reveals the source of the fundamental inefficiency in the C code.

In order to offer a modern object oriented environment, the GObject system requires that a large amount of 'boiler plate' code be written for every class and object. Since the Aldgate prototype Vala sources contain one or more classes per file, the smaller files (those with less methods and functionality) reveal the magnitude of this boiler plate code more readily. For example, each class in the GObject system requires by convention type macros, struct definitions and simple and mostly-repetitive code which offers no useful value to the programmer but requires extra programmer time.

The conclusion from this being that higher level language constructs can serve a real and practical purpose of reducing typing time with programmers, due to their higher 'idea density' - useful ideas per line of code. This has benefits which apply to development, maintenance, refactoring, and general readability.

3.4.1 Caveats

It would be unfair to assume that the C code generated by the Vala compiler is of the same density as that of a skilled programmer, and this is due to the fact that the Vala compiler is optimised for

[4]Reversing Moore's law over the 23 year gap between the creation of C and JavaScript shows that average computer power has increased by 2.9% million.

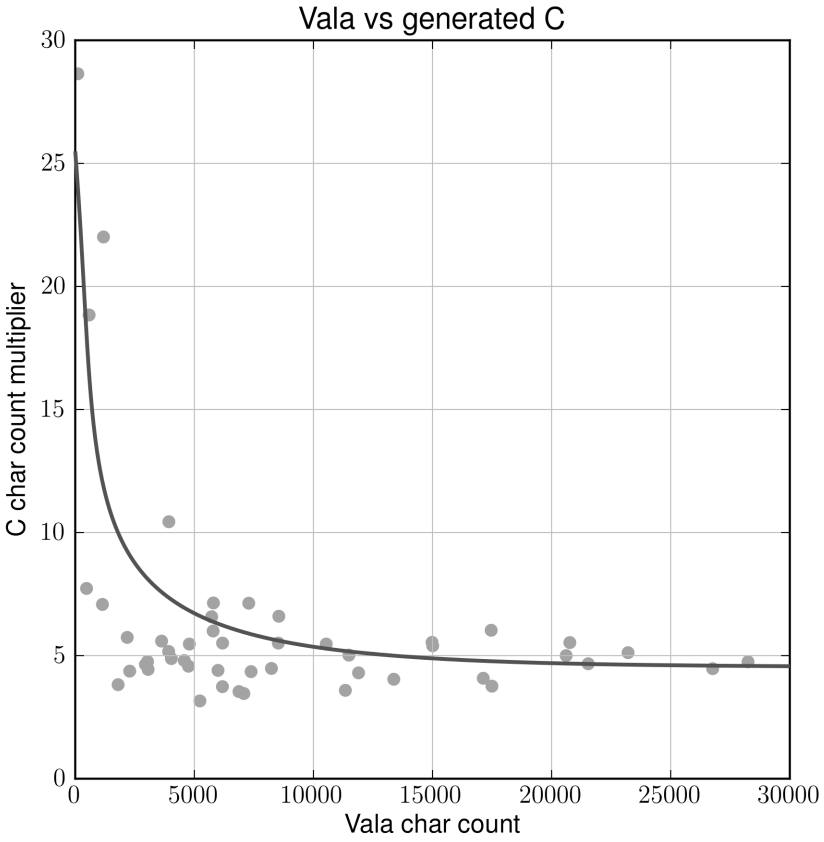

Figure 11: A plot of of non-whitespace character counts for Vala and generated C sources (as an input size multiplier).

integrity rather than code density. This loss of code density has a great influence on simple expressions. A simple integer addition method in Vala translates almost directly into C.

```
public static int add_int (int x,
                           int y)
{
  return (x + y);
}
```

```
gint
foo_bar_add_int (gint x,
                 gint y)
{
  return (x + y);
}
```

Figure 12: An integer addition function in Vala (left), and in C (right).

However, the Vala compiler generates a much more elongated (though functionally identical) snippet:

```
gint foo_bar_add_int (gint x, gint y) {
  gint result = 0;
  gint _tmp0_;
  gint _tmp1_;
  _tmp0_ = x;
  _tmp1_ = y;
  result = _tmp0_ + _tmp1_;
  return result;
}
```

Figure 13: A Vala generated integer addition function.

The above function indicates the simplicity of the Vala lexical analyser, as it shows the function parameters being first transferred to local variables, and the resulting value being calculated before being returned. This is because of simplistic compiler rules whereby a function must return a variable (not an expression, as in Figure 12), and internal expressions may only act upon variables and not parameters (this prevents problems when passing arguments by reference, as they could become inadvertently corrupted by expressions in the method).

Interestingly, a direct comparison of the unassembled code section generated for these two functions shows that the extra instructions present in the Vala generated C function do result do in an increased binary size and extra instructions.

```
add_int:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    -4(%rbp), %edx
        addl    %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
add_int:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    $0, -4(%rbp)
        movl    -20(%rbp), %eax
        movl    %eax, -8(%rbp)
        movl    -24(%rbp), %eax
        movl    %eax, -12(%rbp)
        movl    -12(%rbp), %eax
        movl    -8(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Figure 14: A comparison of handwritten vs Vala compiler generated assembly addition routines. The left column represents the hand written C (Figure 12), whereas the right column represents the compiler generated procedure (Figure 13). Note the extra move (`mov`) instructions which server no purpose other than to rearrange the values of registers.

3.4.2 Results

Despite the drawbacks of the Vala compiler, it is clear from the results of the case study on the aldgate codebase that object orientated programming in C is a more verbose operation than in Vala, and that this extra verbosity presents itself as wasted programmer time, reducing their productivity. Even assuming that a human was able to write the equivalent program in 1/3 of the space that the compiler generated C files consume, there would still be a 24% decrease in codebase size by using Vala over C in the Aldgate project. It is said that good programming is 90% thinking and 10% typing, in which case, the use of Vala over C in these situations would save a company around £2,300 per engineer per year[5]. Such a large return on investment is enough to justify a more serious consideration of programming language choice before starting a new project. Of course this case study is limited in scope only to file size, and doesn't cover the advantages in debugging and other other assets of programming that higher level languages offer.

[5]Calculated with a programmer salary of £30,563 (UK median 2011, source: `http://www.payscale.com/`).

4 COMPUTATION

4.1 Clock Cycles as Free Entities

> *"...we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."* [1]

Just as it was described how the graphical user interface provides a metaphor between the user and the hardware (page 6), abstraction in programming languages provides a metaphor between the programmer and the underlying machine. Instead of having to work exclusively with the native instruction set, programmers have the freedom to construct data structures of their own devising and define operations upon them, leaving the compiler to be responsible for translating them into a machine language. Abelson and Sussman provide an articulate explanation of this in the introduction to their seminal work *Structure and the Interpretation of Computer Programs*, referring to programming languages as a "novel formal medium for expressing ideas about methodology".

It is my belief that in order to achieve this goal when writing code, one must disregard all considerations of performance, since the well meaning aspiration of 'performance optimisation' is responsible for the majority of the gravest cases of unnecessary complexity in programming. The process of optimising code can be considered the opposite of refactoring. While both refactoring and optimising do not affect the work done by a section of code, refactoring it is for the benefit of the programmer, whereas optimising is for the benefit of machine, usually at the programmer's expense. Optimisations almost always obfuscate the code, leading Knuth to famously state "Premature optimization is the root of all evil" [20]. Beck offered a similar interpretation - "Make it run, make it right, make it fast" [4]; both philosophies discourage optimisation until the entire implementation is completed.

As an example of code obfuscation through optimisation, consider this neatly abstracted code segment in which the operation `jumble_values()` is called a number of times over values within two arrays, `a` and `b`:

```
void
jumble_values (int x,
               int y)
{
   x++;
   y += x;
   x = y*x;
}

void
main ()
{
  for (i = 1; i <= 5; i++)
    jumble_values (a[i], b[i-1]);
}
```

Figure 15: A well abstracted code segment.

A logical optimisation for this would be to inline the `jumble_values()` function into the for loop, so as to eradicate the cost of multiple function calls (the cost being increased stack activity, saving/restoring memory addresses etc). A further optimisation could be gained from unrolling the loop, since the number of iterations is fixed and known. The resulting code is shown in Figure 16.

```
void
main ()
{
  a[1]++;
  b[0] += a[1];
  a[1] = b[0]*a[1];
  a[2]++;
  b[1] += a[2];
  a[2] = b[1]*a[2];
  a[3]++;
  b[2] += a[3];
  a[3] = b[2]*a[3];
  a[4]++;
  b[3] += a[4];
  a[4] = b[3]*a[4];
  a[5]++;
  b[4] += a[5];
  a[5] = b[4]*a[5];
}
```

Figure 16: An optimised version of the code segment in Figure 16.

The modifications have obfuscated the meaning of the code by removing the descriptive function name, and the loss of abstraction has led to code duplication, making maintenance harder. For example, a modification to the `jumble_values()` algorithm would need applying identically to each of the five repetitions, violating the Don't Repeat Yourself (DRY) principle.

4.2 On Programming Language Design

> *"A programming language servers two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done."* [34]

A popular view in software engineering is that it is not the language that matters, but the code. This is encouraging the idea that programmers should not get too hung up on language debates (C++ vs Java, etc) or intimidated by the differences in languages, but should instead focus on the code and algorithms themselves. While I believe that this is a well intentioned maxim, it is patently untrue that the programming language choice is irrelevant when programming, and the influence that the choice of programming language makes on a programmer's mindset and problem solving is not to be underestimated.

A programming language is a tool with which to create software, and so indifference towards the tool means indifference towards the very means of creation.

While it is true that algorithms and data structures are abstractions of equal value no matter what language they are implemented in, it would be wrong to assume that this means that all languages are equally well suited to particular tasks. Just as in natural languages, a programming language is a product of environment.

4.3 Case Study B

As a practical example of the differences in language design goals, Figures 17 and 18 contain two different implementations of the quick sort algorithm for list sorting.

```
(define (quicksort list compare-func)
  (define pivot (lambda (list)
                  (cond ((null? list) 'done)
                        ((null? (cdr list)) 'done)
                        ((compare-func (car list) (cadr list)) (pivot (cdr list)))
                        (#t (car list)))))

  (define partition (lambda (piv list p1 p2)
                      (if (null? list) (list p1 p2)
                          (if (< (car list) piv)
                              (partition piv (cdr list) (cons (car list) p1) p2)
                              (partition piv (cdr list) p1
                                         (cons (car list) p2))))))

  (let ((piv (pivot list)))
    (if (equal? piv 'done) list
        (let ((parts (partition piv list () ())))
          (append (quicksort (car parts))
                  (quicksort (cadr parts)))))))
```

Figure 17: A Scheme implementation of the quicksort algorithm.

This example demonstrates several key features of the Scheme programming language. being a LISP dialect - LISP itself being a contraction of **LIS**t **P**rocessing, it offers a generic means for sorting a list of items `list` using a comparison procedure `compare-func` and is recursive in nature. It highlights various properties of the language, such as procedures as first class citizens and function scoping (defining a function within a function). Scheme's simple syntax gives it an intuitive feel, with the most unintuitive aspect of the language being the use of *Polish prefix notation*, wherein the operator is given to the left of the operands i.e. `(= 10 (+ 5 5))` instead of `((5 + 5) = 10)`. This quicksort implementation would be called with `(quicksort '(3 8 5 4 8 2 4 1 9 4))`.

```
#define  MAX_DEPTH 1024

void
quick_sort_int (int *A,
                int n)
{
  int beg[MAX_DEPTH];
  int end[MAX_DEPTH];
  i = 0;
  int piv;
  int left;
  int right;

  beg[0] = 0;
  end[0] = n;

  while (i >= 0)
    {
      left = beg[i];
      right = end[i] - 1;

      if (left < right)
        {
          piv = A[left];

          if (i >= MAX_DEPTH - 1)
            return; /* Max depth reached. */

          while (left < right)
            {
              while (A[right] >= piv && left < right)
                right--;

              if (left < right)
                A[left++] = A[right];

              while (A[left] <= piv && left < right)
                left++;

              if (left < right)
                A[right--] = A[left];
            }

          A[left] = piv;
          beg[i++] = left + 1;
          end[i] = end[i];
          end[i] = left;
        }
      else
        i--;
    }
}
```

Figure 18: An implementation of the quicksort algorithm in the C programming language.

The C implementation in Figure 18 shows some of the key properties of the language. It offers a highly optimised algorithm, using iteration over recursion - at the expense of robustness. While both implementations achieve the same goal, they demonstrate several differences in the designs of the Scheme and C programming languages. C's strongly typed object model stands in contrast to Scheme's untyped model, meaning that the C implementation is restricted to sorting integer arrays 'A' of size 'n', whereas the Scheme implementation will accept a list of any type for which there is a valid a `compare-func`. Additionally, Scheme's automatic memory management makes the distinction between passing function parameters by reference or value meaningless, as opposed to C flat memory model.

4.3.1 Results

In the above example, the properties of the language have had a strong influence on the 'flavour' of the implementation. The Scheme implementation offers a generic, robust and abstract means for sorting data, whereas the C implementation provides an lean and incredibly efficient method for ordering sets of integers. While it is not impossible to implement either version in both languages, the language provides the environment in which the programmer solves problems, and so has a direct and strong influence on the way in which the solutions are approached.

4.4 On Cognition in Programming

In recent years there has been a large amount of research and development put into the optimisation stage of compilers, to the extent where the codebase of a modern compiler can be expected largely to be dominated by optimisation rules as opposed to actual lexical analysis or code generation components.

This is indicative of a common trend - that of cognitive offloading from the programmer to the compiler. For maximum efficiency, a programmer should be focused only in the immediate task at hand, and compiler-side optimisations offer a similar a form of cognitive easing by enabling low level performance and instruction decisions to be made by the compiler, thus alleviating the programmer of the responsibility of writing code with consideration for the target hardware.

An additional benefit of allowing the compiler rather than the programmer to perform optimisations is that it allows better specialisation of skills, as the authors of compilers can specialise in understanding performance considerations of the target hardware, allowing the programmer to focus on the task at hand. This means that compilers can be written that perform better optimisations than programmers. For example, in the case of loop unrolling optimisations, only a programmer with a deeply intimate knowledge of the target hardware would understand when the increased cost on the processor's instruction cache renders a loop unroll to be a poor optimisation choice. On the other hand, a table of instruction cache sizes could easily be stored in the optimisation section of a compiler, allowing it to make compile time optimisation decisions.

"Modern C and C++ compilers do excellent optimization... The compiler writers understand the CPU architecture quite well." [33]

4.5 Case Study C

Unlike an assembler which is essentially just a macro processor, an optimising compiler has choices over what machine code to generate. By offloading the responsibility of optimisations from the programmer to the compiler, the compiler has an increased over the machine code that gets compiled. This means that there is no guarantee that binaries generated form the same source with different compilers will match, and this can be demonstrated using a simple exercise in C (Figure 19).

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

int
main ()
{
  int i, x;
  double y, z;

  x = rand () % 8;
  y = pow (x, 2.0);
  z = y * 52;
  printf ("%d%f", x, y);

  for (i = 0; i < MAX; i++)
    {
      int a;

      a = i * 12;
      printf("%d", a);
    }
  return 0;
}
```

Figure 19: A simple C program to test compiler side optimisations.

To demonstrate the opportunities for compilers to produce differing machine code, the GNU GCC and Clang C compilers will be used. The GNU GCC C compiler is GNU's implementation of the UNIX CC compiler, and is a very mature and widespread project, being the most common open source C compiler. Clang uses an LLVM infrastructure and is a younger project, being developed largely by Apple and Google.

Assembly files were generated by both compilers using the same configuration, and the output compared.

```
$ gcc -O3 -S optimise-me.c -lm
$ mv optimise-me.s optimise-me-gcc.s
$ clang -O3 -S optimise-me.c -lm
$ mv optimise-me.s optimise-me-clang.s
```

4.5.1 Results

While there are superficial differences in instruction order and register choice between the two compilers, they both produce essentially the same proceedures, unrolling the for loop into a linear list of instructions, and precomputing the results for the `a = i * 12` calculation, replacing it with the values 12...144. This style of keyhole optimisation is a common and effective way of reducing the processor time required for an operation, allowing as much computation as is possible to be performed at compile time.

GCC

```
movl $12, %esi
movl $.LC1, %edi
xorl %eax, %eax
call printf
```

Clang

```
movl $.L.str1, %edi
movl $12, %esi
xorb %al, %al
callq printf
```

Figure 20: Assembly code for a single iteration (where `i = 1`) of the unfolded loop generated from Figure 19.

However, the differences between the two compilers' approach to loop unrolling is revealed when the value of `MAX` is adjusted. If `MAX` >17, the GCC compiler will decide that an inner loop is a more effective manner and will collapse the loop, however the Clang compiler will continue to unroll the loop all the way until `MAX` >21. This represents a 20% difference in maximum loop size, highlighting a difference in decision making between the two sets of developers.

The more that optimisation can be handled by the compiler, the less responsibilities the programmer has, giving the obvious benefit of reducing code complexity through lesser optimisations.

4.6 Case Study D

One of the largest areas of development in modern language design has been in that of memory management. From automatic allocation through to garbage collection algorithms, a lot has been done to abstract the direct management of memory away from the programmer and into the language. One of the means by which this is achieved is by adding memory protection.

In flat memory model languages like C, there is no means of protecting a user form invalid memory access. For example, a program can index outside of an array, with undefined results - at best, the kernel will detect invalid memory access and kill the process with a Segmentation Fault, at worst, the invalid access will go unnoticed and perhaps corrupt the memory of other processes.

To demonstrate bounds checking, consider the following trivial function in C to index into an integer array `A` by index `x`, returning a pointer to the element.

```
int *
access_array (int *A,
              int  x)
{
  return &A[x];
}
```

```
access_array:
        movslq  %esi, %rsi
        leaq    (%rdi,%rsi,4), %rax
        ret
```

Figure 21: Array access function in C (left) and the corresponding assembly routine (right).

To add bounds checking, additional checks must be made to ensure that $0 \leq x < n$, where n is the size of array A. This makes the last legal access `A[n-1]`. Figure 22 shows an implementation of this bounded array access, using a ternary operator and a conditional branch in the assembly routine.

```
int *
access_bounded_array (int *A,
                      int  n,
                      int  x)
{
  return (x >= 0 && x < n) ? &A[x] : NULL;
}
```

```
access_bounded_array:
        cmpl    %esi, %edx
        jge     out_of_bounds
        testl   %edx, %edx
        js      out_of_bounds
        movslq  %edx, %rdx
        leaq    (%rdi,%rdx,4), %rax
        ret
out_of_bounds:
        xorl    %eax, %eax
        ret
```

Figure 22: Bounded array access C function (left) and assembly routine (right).

The compare (`cmp`), conditional jumps (`jge` and `js`), and logical AND (`test`) instructions add an additional 4 instructions to the array access execution routine.

4.6.1 Results

While CPU design is at a level of complexity (with out of order execution, pipe-lining etc) whereby it is not possible to calculate a definite execution time for a given set of instructions, the addition of bounds checking to the array access routine results in a 133% increase in the number of executable instructions. Assuming each execution executed in equal time, this would result in a 57% decrease in execution speed for the bounded array access. Of course, optimisations can be made to reduce this number (such as performing the bounds check once on a sequence of array accesses), but this does demonstrate the increased expense of memory protection.

Additionally, this bounds checking implementation means that calling routines must test for `NULL` return after each call, meaning that there is an extra layer of conditional code. This would make bounded array access very cumbersome, which is why many modern languages use an Exception based error type, whereby an invalid array access would throw an Exception detailing what went wrong, which would either provide diagnostic debugging information or would be caught and gracefully handled by the calling code.

This demonstration of the expense of memory protection helps explain why C and other low level languages are still used in performance critical situations such as in the kernel and in embedded systems.

5 TRANSPARENCY

Transparency in software can be characterised in three forms: transparency of development, transparency in applications, and transparency of code. Transparent of development means clarity about the process, motivation and personnel of a project. An application can be considered transparent if it has a clearly defined purpose, interface and behaviour. Transparent code is code that is easily read and understood by other developers.

It is my contention that maintaining transparency among these three areas offers a great tool in managing complexity in application development; and they are tightly interlinked - without transparency of development, there can be no transparency in applications; without transparency of code, there is no transparency of applications.

5.1 Transparency of Code

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."* [12]

To achieve transparency of code, standards for writing must be adopted, as familiarity is born of uniformity. An often used tool in achieving uniformity of code is the Coding style - a plaintext document which describes the layout and format of code for a given scope. Examples of this include the Oracle Code Conventions for the Java Programming Language and the GNU Coding Standards. Distributing such a document is a convenient way to inform others of the style of code being used, however it does not guarantee adherence. A code standard only has as much value as its users assign to it, and so participation in a code style is voluntary. When writing a program, the compiler will fail if there is an error in the use of the target language; there is no 'grey area' - code either compiles or doesn't. The fact that a code style requires voluntary behaviour means that it can never achieve this level of certainty. Where a human is involved, there is room for interpretation, and so no two people may agree on what constitutes 'clean code', even though they could both identify 'correct code'.

5.2 Transparency of Development

In recent years, the Open Source movement has entered common language and has achieved a great amount of attention from developers and users. From Raymond's 'Cathedral & the Bazaar' essays through to the popularisation of Google's Android operating system, it is becoming apparent that as a software development process, the release early, release often model [30] has significant merit.

One of the effects of the popularisation of personal computing is that everyone can be an expert. The constant exposure to computing that society has created means that decisions on what makes appropriate software are no longer the sole realm of the academic, or the software developer. Now more than ever, a lot can be learned by drawing an analogy to what Da Vinci considered to be the greatest way of painting. The analogy to software development is painfully self-evident so I will end this section with an excerpt from his notebooks which details the methodology.

> *"Certainly while a man is painting he ought not to shrink from hearing every opinion. For we know very well that a man, though he may not be a painter, is familiar with the forms of other men and very capable of judging whether they are hump-backed, or have one shoulder higher or lower than the other, or too big a mouth or nose, and other defects; and, as we know that men are competent to judge of the works of nature, how much more ought we to admit that they can judge of our errors' since you know how much a man may be deceived in his own work. Therefore be curious to hear with patience the opinions of others, consider and weigh well whether those who find fault have ground or not for blame, and, if so amend; but not make as though you had not heard, or if he should be a man you esteem show him by arguments the cause of his mistake."* [7]

6 CONCLUSIONS

It is my contention that choice and use of programming language is the single most under-appreciated tool for controlling software complexity, and in envisioning the different approaches to complexity management, it can be helpful to entertain the thought experiment of a 'perfect programming language' - that is, a language in which there is no semantic gap between the the code that is written and the idea that motivated it. In this language, an idea can be explicitly communicated using a single statement, and these statements can be compounded into more complex operations with absolutely no possibility for ambiguity or differences in interpretation.

While of course unattainable, such a goal does reveal useful insights into the nature of language design. The first is that it would eradicate the need for coding styles, since their purpose is to limit a programmer's use of language to a common uniform level (i.e. where there a two ways of achieving a goal, a code style will specify which to use). This then leads to the subsequent conclusion that in the perfect language, there would only ever be a single method of achieving a goal. In doing so, the programmer loses the ability to control *how* a program will execute, meaning that there will be no connection between the source code and the machine code at all, relying on pure abstraction provided by the compiler. So in order to achieve higher levels of abstraction, a programmer must sacrifice power over the hardware in favour of increased efficiency of communicating ideas.

To this end, the use of very low level languages is likely to remain consistent in areas of software where there is no desire to communicate ideas, such as in kernel development - since there is no need to make this sacrifice in control over efficiency.

It is my prediction that as the size and interactivity of applications continues to grow, we will see continued expansion in the number and use of higher level and domain specific languages in an open source environment, with an increasing number of developers willing to trade absolute control over their processes for the ability to produce increasingly interactive applications in increasingly shorter time-spans.

A Intel Transistor Density

Processor	Year	Transitors
4004	1971	2,300
8008	1972	2,500
8080	1974	4,500
8086	1978	29,000
Intel286	1982	134,000
Intel386TM processor	1985	275,000
Intel486TM processor	1989	1,200,000
Intel Pentium processor	1993	3,100,000
Intel Pentium II processor	1997	7,500,000
Intel Pentium III processor	1999	9,500,000
Intel Pentium 4 processor	2000	42,000,000
Intel Itanium processor	2001	25,000,000
Intel Itanium 2 processor	2002	220,000,000
Intel Itanium 2 processor (9MB cache)	2004	592,000,000

Table 1: Number of transistors in Intel microprocessors over time [17].

B GitHub Language Popularities

Language	Year	%
C	1972	6
C++	1983	4
Objective-C	1983	3
Perl	1987	4
Python	1991	8
PHP	1995	7
Java	1995	8
Ruby	1995	13
JavaScript	1995	21
Shell		8
Other		18

Table 2: Popularity of languages in pulic GitHub repositories [14].

C Case Study A Results and Methodology

The following bash script was used to list the size of input vala source files against their compiler generated C source counterpart. Each vala source file is compiled to a C source file with matching file name. The program prints a tabular list of vala-c pair values in the following format:

```
<line-count> <word-count> <character-count> <basename>
```

Where `basename` is the shared component of the two source file names, stripped of the project prefix. Figure 23 demonstrates this process.

Vala: `vala/aldgate-activity-manager.vala`
C: `generated/aldgate-activity-manager.c`
Basename: `activity-manager`

Figure 23: Aldgate file path extraction from Vala source, generated C and basename.

The final line of output is the total counts for each pair value.

```
#!/bin/bash

# aldgate-protoype-wc
# Chris Cummins - 8th December 2012
# Compare word line and char counts for aldgate vala sources with generated c files.

total_vala_cc=0; total_c_cc=0; total_vala_wc=0; total_c_wc=0; total_vala_lc=0;
total_c_lc=0; dir=~/src/aldgate-prototype/jni

count ()
{
    echo $(wc -$2 "$1" | awk '{ print $1 }')
}

for file in $(ls $dir/vala)
do
    base="${file%.vala}"
    vala_file="$dir/vala/$file"; c_file="$dir/generated/$base.c"
    vala_lc=$(count "$vala_file" l); c_lc=$(count "$c_file" l)
    vala_wc=$(count "$vala_file" w); c_wc=$(count "$c_file" w)
    vala_cc=$(count "$vala_file" c); c_cc=$(count "$c_file" c)
    lc=$(echo "$c_lc / $vala_lc" | bc -l | xargs printf "%.1f")
    wc=$(echo "$c_wc / $vala_wc" | bc -l | xargs printf "%.1f")
    cc=$(echo "$c_cc / $vala_cc" | bc -l | xargs printf "%.1f")
    total_vala_lc=$((total_vala_lc+vala_lc)); total_c_lc=$((total_c_lc+c_lc))
    total_vala_wc=$((total_vala_wc+vala_wc)); total_c_wc=$((total_c_wc+c_wc))
    total_vala_cc=$((total_vala_cc+vala_cc)); total_c_cc=$((total_c_cc+c_cc))
    echo -ne "$vala_lc\t$c_lc\t$lc\t$vala_wc\t$c_wc\t$wc\t$vala_cc\t$c_cc\t"
    echo -e "$cc\t${base:8}"
done

avg_lc=$(echo "$total_c_lc / $total_vala_lc" | bc -l | xargs printf "%.1f")
avg_wc=$(echo "$total_c_wc / $total_vala_wc" | bc -l | xargs printf "%.1f")
avg_cc=$(echo "$total_c_cc / $total_vala_cc" | bc -l | xargs printf "%.1f")
echo -ne "$total_vala_lc\t$total_c_lc\t$avg_lc\t$total_vala_wc\t$total_c_wc\t"
echo -e "$avg_wc\t$total_vala_cc\t$total_c_cc\t$avg_cc"
```

Figure 24: `aldgate-wc` Bash script.

Results

```
 $ valac --version
Vala 0.16.1
 $ ./aldgate-prototype-wc > ~/dissertation/res/aldgate-prototype-wc-results
 $ cat aldgate-prototype-wc-results

353     1683    4.8     885     5320    6.0     10538   57690   5.5     activity-manager
54      630     11.7    138     1892    13.7    1185    26087   22.0    activity
391     1729    4.4     927     5490    5.9     11489   57658   5.0     application-manager
557     2120    3.8     1214    6633    5.5     14918   80838   5.4     application-switcher
163     826     5.1     397     2632    6.6     4790    26205   5.5     application
700     2914    4.2     1732    9188    5.3     20610   103051  5.0     apps-list
290     1261    4.3     637     4123    6.5     7285    51966   7.1     apps-screen
380     1352    3.6     876     4316    4.9     11895   51132   4.3     app-tile
100     417     4.2     240     1370    5.7     2947    13676   4.6     clock
201     958     4.8     405     3002    7.4     5727    37677   6.6     comms-contact-details
642     2804    4.4     1610    8798    5.5     20766   114870  5.5     comms-contact-history
291     911     3.1     623     2713    4.4     8237    36942   4.5     comms-contact-list-item
569     2669    4.7     1275    8446    6.6     17463   105385  6.0     comms-contact-list-panels
285     1250    4.4     743     3968    5.3     8525    47013   5.5     comms-contact-list
204     872     4.3     415     2689    6.5     5789    34745   6.0     comms-contact-scroll-view
542     1877    3.5     1252    6022    4.8     17391   71010   4.1     comms-contact-sms
119     785     6.6     289     2600    9.0     3672    38305   10.4    comms-screen
143     549     3.8     338     1777    5.3     3921    20257   5.2     confirm-menu
782     3272    4.2     1760    10379   5.9     23210   118734  5.1     contact-editor
99      384     3.9     235     1192    5.1     3059    13575   4.4     contact-list-picture
503     2476    4.9     1293    8016    6.2     14980   82939   5.5     contact-manager
181     712     3.9     491     2203    4.5     6874    24341   3.5     contact-picture
634     2299    3.6     1583    7174    4.5     17495   65720   3.8     contact
174     509     2.9     533     1578    3.0     5233    16557   3.2     deform-box-layout
201     729     3.6     622     2258    3.6     6180    23114   3.7     deform-grid
124     357     2.9     267     1175    4.4     3420    15222   4.5     disclosure-clipper
167     571     3.4     361     1854    5.1     4755    21708   4.6     dock
154     587     3.8     327     1839    5.6     4035    19682   4.9     header
50      219     4.4     100     679     6.8     1141    8083    7.1     home-button
36      308     8.6     68      965     14.2    583     10984   18.8    item-group
684     2290    3.3     1616    7291    4.5     21537   100671  4.7     lock-screen
242     725     3.0     656     2248    3.4     7079    24468   3.5     main-box-layout
67      264     3.9     187     903     4.8     2286    9981    4.4     main-scroll-view
166     629     3.8     403     1970    4.9     4579    21972   4.8     menu-header
198     879     4.4     451     2809    6.2     5760    41338   7.2     notification-banner
478     1416    3.0     1074    4481    4.2     13374   54025   4.0     notification-box
204     812     4.0     464     2592    5.6     7383    32131   4.4     notification-client
831     2954    3.6     2376    9392    4.0     26653   119146  4.5     notification-screen
132     521     3.9     313     1596    5.1     3625    20277   5.6     notification-tile-list
955     3353    3.5     2384    10523   4.4     27501   132473  4.8     notification-tile
334     1410    4.2     740     4376    5.9     8547    56399   6.6     notification
67      213     3.2     169     625     3.7     1799    6871    3.8     overlay
87      351     4.0     173     1111    6.4     2182    12517   5.7     panel
244     878     3.6     564     2878    5.1     6184    34043   5.5     screen
18      91      5.1     56      260     4.6     474     3663    7.7     scroll-view
6       74      12.3    15      204     13.6    108     3093    28.6    secondary-button
252     798     3.2     618     2507    4.1     5994    26361   4.4     stack
338     1223    3.6     733     3930    5.4     11341   40737   3.6     statusbar
14392   56911   4.0     34628   179987  5.2     424519  2135332 5.0
```

Figure 25: Resulting output from aldgate-wc.

Results Processing

```
#!/bin/python

import matplotlib
from matplotlib import rc
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
import matplotlib.mlab as mlab
from pylab import *
from numpy import *

rc('text', usetex=True)
r = mlab.csv2rec("cc.csv", delimiter='\t')
v = r[r.dtype.names[0]]
m = r[r.dtype.names[1]]
fig = figure (figsize=(6, 6), dpi=72)
ax = subplot (1, 1, 1)
ax.set_title ("Vala vs generated C", fontsize=14)
ax.set_xlabel ("Vala char count", fontsize=12)
ax.set_ylabel ("C char count multiplier", fontsize=12)
ax.grid (True, linestyle='-', color='0.75')
ax.scatter (v, m, s=20, color='tomato');
xlim (0, 30000)
savefig ("plot.png", dpi=320)
show ()
```

Figure 26: plot.py Python script.

D References

[1] Abelson, H., Sussman, G. J. (1996) Structure and Interpretation of Computer Programs, second edition. MIT Press.

[2] Agarwal, A. (2009) Facebook: Science and the Social Graph, available at: http://www.infoq.com/presentations/Facebook-Software-Stack (Accessed: 1st December 2012).

[3] ARM Holdings (2012) 2011 Preliminary Results, available at: http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9MTI0NjY3fENoaWxkSUQ9LTF8VHlwZT0z&t=1 (Accessed: 8th December 2012).

[4] Beck, K. (1997) Make It Run, Make It Right: Design Through Refactoring. Volume 6, Issue 4 of SIGS Publications.

[5] Brooks, F. P. (1995) The Mythical Man-Month, anniversay edition. Addison-Wesley: University of North Carolina at Chapel Hill.

[6] Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009) Introduction to Algorithms, third edition. MIT Press.

[7] Da Vinci, L. (2005). Da Vinci Notebooks. Profile Books.

[8] Dijkstra, E. W. (1968). Go To Statement Considered Harmful.

[9] ElectronicsWeekly.com (2010) Samsung to overtake Intel as No.1 chip company in 2014, available at: http://www.electronicsweekly.com/Articles/26/08/2010/49337/samsung-to-overtake-intel-as-no.1-chip-company-in-2014.htm (Accessed December 4th 2012).

[10] Financial Times (2012) Intel and Qualcomm step up chip battle, available at: http://www.ft.com/cms/s/2/828b38dc-3be1-11e1-82d3-00144feabdc0.html (Accessed: 8th December 2012).

[11] Forbes (2012) Intel Should Be A $33 Stock As The Market Overlooks Its Dominance, available at: http://www.forbes.com/sites/greatspeculations/2012/11/30/intel-should-be-a-33-stock-as-the-market-overlooks-its-dominance/ (Accessed: 8th December 2012).

[12] Fowler, M. (1999). Refactoring: Improving the Design of Existing Code, first edition. Addison Wesley.

[13] Gauthier, R., Pont, S. (1970). Designing Systems Programs.

[14] GitHub (ND) Top Languages, available at: https://github.com/languages (Accessed: 23rd December 2012).

[15] GovernmentIT (2009) NASA uses Unix to track asteroid's close call to Earth, available at: http://www.computerworld.com/s/article/9128975/NASA_uses_Unix_to_track_asteroid_s_close_call_to_Earth (Accessed: 1st December 2012).

[16] Hoare, C. A. R. (1981). The Emporeror's Old Clothes.

[17] Intel Corporation (2005) Moores Law Backgrounder, available at: http://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_Backgrounder.pdf (Accessed: 2nd November 2012).

[18] Intel Corporation (2012) Financials and Filings, available at: http://www.intc.com/financials.cfm (Accessed: 8th December 2012).

[19] Kerningham, B., Pike, R. (1976) Software Tools. Addison Wesley.

[20] Knuth, D. (1974) Structured Programming With Go To Statements. Computing Surveys, Volume 6, Number 4.

[21] The Linux Foundation (2010) Public Support for the MeeGo Project, available at: `http://www.linuxfoundation.org/node/6144` (Accessed: 1st December 2012)

[22] Martin, R. C. (2008) Clean Code: A Handbook of Agile Software Craftsmanship, first edition. Prentice Hall.

[23] Martin, C. (2011) Agile Software Development, Principles, Patterns, and Practices, first edition. Pearson.

[24] McCabe, T. J. (1976) A Complexity Measure. IEEE Transactions on Software Engineering: Vol. SE-2, No. 4.

[25] Miller, G. A. (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review Vol. 101, No. 2, 343-352, American Psychological Association.

[26] Moore, G. (2003) International Solid-State Circuits Conference (ISSCC).

[27] Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules.

[28] Patterson, D. A., Ditzel, D. R. (1980) The case for the reduced instruction set compiler.

[29] phoneArena.com (2012) Qualcomm continues to dominate mobile market while Intel nibbles 0.2% share, available at: `http://www.phonearena.com/news/Qualcomm-continues-to-dominate-market-while-Intel-nibbles-0.2-share_id35252` (Accessed: 8th December 2012).

[30] Raymond, E. S. (2001) The Cathedral & the Bazaar, first edition. O'Reily Media.

[31] Raymond, E. S. (2003) The Art of Unix Programming, first edition. Addison Wesley Professional Computing.

[32] Samsung (2011) Samsung Profile 2011, available at: `http://www.samsung.co.kr/samsung/outcome/perfomance.do` (Accessed: 8th December 2012).

[33] Seyfarth, R. (2011) Introduction to 64 Bit Intel Assembly Language Programming for Linux. CreateSpace Independent Publishing Platform.

[34] Stroustrup, B. (1997) The C++ Programming Language, third edition. Addison Wesley.

[35] Tognazzini, B. (1992) Tog on Interface. Addison Wesley.

[36] Turing, A. M. (1936) On Computable Numbers, with an Application to the Entscheidungsproblem.

[37] W^{3}Techs (ND) Usage of operating systems for websites, available at: `http://w3techs.com/technologies/overview/operating_system/all` (Accessed: 9th December 2012)

[38] W^{3}Techs (ND) Usage statistics and market share of Unix for websites, available at: `http://w3techs.com/technologies/details/os-unix/all/all` (Accessed: 6th December 2012)

Word count: