Untrapped Value

Untrapped Value:

This is a series to follow
with no ends

Software Reuse Powering Future Prosperity

Dave R. Erickson

EVERGENT TECHNOLOGIES
1.01 GAMMA VERSION $\frac{\$39.99\pi}{\pi+0.0...1}$

Book Design reused from:

For your future prosperity, everywhere, and for everyone!

You Are What You Have Been,
And What You Will Be Is What You Do.
—Siddhartha Gautama, The Buddha

Contents

List of Figures

List of Tables

Macroscopic View: The Manifesto

Chapter 1

Introduction

Since Everything is a Reflection of Our Minds,
Then Everything Can Be Changed By Our Minds.
—Siddhartha Gautama, The Buddha

Mankind has invested vast resources (time, manhours, computer machinery sunk costs, maintenance, building space, heating, venting, cooling, and so on) into software for all kinds of digital and analog hardware for over sixty years. Far longer if you consider punched cards, and so on. In the end, most of the source code ends in the waste heap of history. Old code gets forgotten, rubbished, and a new wave of developers is forced to recreate new versions of old ideas. People get promoted, graduate from college, and leave to get married; before they do they don't have time, don't believe in the priority, and don't place the code where others can find it to make an important curation of their software; and by this donate it to future generations, worldwide, the society at large. If organizations, at the other end of the spectrum, would realign software for a legacy of centuries instead of product runs, mankind can preserve the sunk costs, speed up advancement, and make software impact far wider when it's made in a reusable form. People move to a new job, and remake linked lists, factory classes, or ring buffers in the new language of the day, or within the design paradigm of the latest fad management.

It's kind of insane when you think about it, people spend many years getting a consumer product working, finely tuned and profitable. Then two companies merge, product lines are unified or obsoleted, and some or all of the intellectual property gets forgotten in a corner as one team is merged and the others retire to golf, or the pool. While filling in cardboard boxes of stuff as they leave, does anyone drag out the old tapes and floppies to make sure the new guys aren't starting by reinventing the wheel?

Why? The culture has a serious misunderstanding of where the value, where the intellectual property comes from and where it gets stored. This wasteful malaise needs to change.

This book is a launching point, not a destination. It is designed to evolve in small, incremental ways along with your reusable software development guidelines, over many years. From novice coders starting out to experienced, and jaded, software managers; all practical and technical issues are presented in two natural layers (for the simplest stratifications - explained in Section 8)- one, the manifesto paints broad strokes in a proscriptive manner about how to steer your organization gradually towards code for longevity, and two, the toolbox brings together a set of free tools to get you started, a bunch of tried and true realities about what makes sense while plumbing inside someone else's code, and realistic high level strategies to make sense of what you find. There's no practical way for this small book to cover every topic fully, the manuals alone for autotools are several thousand pages. But the goal is a comprehensive perspective, and that can be achieved, quickly.

This book provides a wider perspective, by looking back on the history of software reuse, and the development cycle not as a painful target to meet and then forget, but as a stepping stone that brings on differing teams, ramping up and ramping down, to meet the custom needs of every stage of software. Doesn't that sound more productive, on the face of it?

Maintenance was the old end of software development, the goal of software reuse is to make all software a continual maintenance cycle for mankind. The goal is to accelerate the next generation farther and faster, perhaps into the stars. But even in a humble grounded form, make impacts felt worldwide.

Easier to start, longer to impact, cheaper to deliver. The goal of all software reuse: to untrap all the value stored there by society.

For each chapter, I parallel the main ideas of reuse with a Buddha koan. Enigmatic ideas smashed together like koans are like the perfect proscriptive advice: they present paradoxical and enigmatic ideas that appear ungrounded in the importance of the day, until you wander into a situation, perhaps a conflict of ideas, and the answer leaps out of the confusion - linked to your brain by the wise words of a koan. It all becomes clear, with time, patience, and practice.

Like the discipline needed to transform people's habits to instill software reuse, Buddhism is a practice. It is a "life raft built for one", as the expression goes. There are many aspects, many dimensions, to consider as important factors in making software more useful to a wider group. Like any discipline, there will be areas people stumble, and other areas where people excel, and areas that take a great deal of resources to conquer. And conquer them you shall, with some humble guidance and a positive outlook.

Executive Summary:

What is this book about?

Software, made with quality from original sources repurposed, in many agreed standpoints of comprehension, to meet a wider audience that benefits mankind for generations instead of fiscal quarters so mankind can maximize benefit from it for all society.

Who may benefit?

Mankind should be interested in and profit from software reuse, because reducing software development time reduces energy greenhouse gas emissions, reduces computing machinery wear and tear, provides more ways to accelerate more people to work on software with security, mission-critical, and real-time requirements; it provides easier starts for younger scientists and engineers in Science, Technology, Engineering, and Math (STEM) to profit from and accelerate their learning and contribution to technology.

Why should society care?

Society has learned from enough bad ideas and bad methods in the progeny of software to make optimization a priority for everyone's advancement tomorrow.

When?

The change needs to happen tomorrow, and this book points a way towards it.

How?

By attacking the top and the bottom of software development at the same time: the first half of this book describes the ideas from a managerial, or high level perspective; the second half delves into the nuts and bolts things anyone might use to get started.

In this chapter we set out the grand visions of what software reuse might be, in the next we bring forward the ideas that will be top of mind if you venture into the business of software reuse.

Chapter 2

Software Reuse: Conceptually

With Our Thoughts,
We Make The World.
—Siddhartha Gautama, The Buddha

Software is a collection of ideas. Whether in human readable thoughts like a book, or in machine language code in many files, or stored on firmware, even the relay-ladder logic in a garbage truck's electrical relays turning hydraulics on and off, these ideas outlive the form they are encapsulated in. That is, they are built to outlast the implementation, but mankind hasn't valued idea capture to the same priority as productivity or novelty.

In the time of hectic progress, we tend to narrow our view from the horizon down to the next corner, the next obstacle to avoid, then next fiscal quarter. That doesn't foster the kind of higher ideals that come with thinking of keeping software like fine artwork. Leonardo da Vinci may have taken 5 years to paint the *Salvator Mundi*, but that still amounts to five man-years. The auction value of *Salvator Mundi* was $497.8 million USD. Many software projects take man-decades to produce at valuations that are nowhere near as valued. Perhaps the mindset must start at appreciating software value in bigger terms?

Take a step back from your problems, from your recent mindset, and consider the age of some ideas.

Sir Isaac Newton's Principia Mathematica was written in 1687: 5 July, 1687. The world's application of these mechanical laws were dovetailed into mechanical machines over generations, the cotton ginny, the steam engine, and so on. Some of the greatest uses of Newton's Laws of Motion arrived with generators and engines of the 18th and 19th centuries. Some would argue that the greatest application was achieved by Orville and Wilbur Wright, bicycle mechanics from

Ohio, when they achieved powered flight[1] on December 17, 1903. What might have happened if Newton's Laws died out in books and devices, in the 17th century?

That's a time difference of about 216 years from the publishing of Newton's Laws to a heavier-than-air machine powered flight. A success for mankind that makes the full cycle from knowledge discovery, through knowledge innovation, unto technological implementation that impacts us all.

Now, consider how efficient mankind might be if the novel ideas in circulation amongst the thought leaders of the day, and the people that innovate upon them, if they were created, operated, and then left in the dustheap of history within a few months. Or a few years, perhaps a few decades? The duration of an idea is ENTIRELY DEPENDENT on the media it inhabits, including the minds of those that understand it. Not the relative applicability of current or over the horizon applications. If it is kept in the catalog, if it gets recycled, it lives on long enough to make it to the next great use. Unfortunately, the final location of many ideas inhabits two vastly different outcomes: ideas that win over the marketplace - by profit margin alone - live on, and the ideas that lose the marketplace in the short term, become detritus. Ideas that may not have arrived "on time" may be just the solution people need just a few years later. Will those ideas be there when it counts? Are we aiming too low to our grandkids' detriment?

Society has many needs. And it has organizations with corporate memories measured in centuries because they are entrusted with maintaining the societies they oversee. The human justice system, the medical healthcare system, politics, technology-based societies, and so on. The human endeavours of STEM, the cradle of all technologies and the lifeblood of tomorrow's needs, must have access to all ideas. Including vacuum tubes. Including Van De Graaf generators. Including hydraulic swash plates.

The obvious retort would be to question why the ideas have to be put back into open software when they have been in software before? And don't they exist in many books, papers, journals, and so on? Well, the first problem is that admits massive failure to keep value. How many wasted efforts would be enough to satisfy anyone's needs? Second, if the result of quickly built software code, full of bugs and wished changes, was not optimal then wouldn't it be better for all to expend that effort on improving the existing code to a better standard than the original? Ideas in theory, in books or videos, aren't the

[1] One might argue Bernoulli's effect was more important and it was discovered around 1738. Yes, but I wanted a bigger difference than 100 years to illustrate my point. Please allow me to reuse Newton, instead.

finished product, working software is the finished product. Books don't encapsulate all the problems, bugs like noninitialized pointers surface using only after hardware, firmware, and software are brought together in integrated form. To suggest you don't need the finished software source product is to accept reinventing the felloe's plate (an American wagon wheel is made of a hub, spokes, felloe's, and tires. You need a felloe's plate to hold the felloe's together under the tire) over and over.

And let's talk about climate change and topical wastes of energy like Bitcoin block chain computations. Does it make sense for the environment, for mankind, for political expediency, and so on, to continually pay for new sets of people to reinstate the same software ideas over and over for a new set of applications? In a time of saturated needs, should software be allowed to be as wasteful?

If you know object-oriented software, and have suffered some sort of odd behaviour that comes and goes or after a revision does things you've never seen before, that is an open secret that the objects being used together, mainly by an Object Oriented Programming (OOP) concept known as inheritance, that has changed the obvious behaviour to the user comes from a place of noncomprehension. Object oriented programming, the last bug fad, results in rapid use of objects without necessarily a complete understanding of this version of how the software works after updates, or the last version. Doesn't it seem like the current model has done a poor job of optimizing how people make code if they can't solve the bug and crashes issues after decades of work?

What this book does, what value it delivers to you now and into the future, is that it combines the ideas and the mechanisms together to act from the large and the small perspectives onto the entirety of the problem. There is no way any novella, nor a multi-part encyclopedia will cover all the important reuse aspects to the satisfaction of all. It doesn't bog down at the technical level, but it demonstrates with real economic numbers how fast you can recapture value. It doesn't solve all problems of unseen unknown work, but it presents the necessary concepts and broad strategies that will inspire your thinking on how to augment code for your own use.

Ideas in a useable form: Comparables

Let's begin our analysis by explaining what kind of things software reuse should include and look like if it wants to attain the same stature of usefulness and importance in STEM; by comparing the problems and needs of reusable software to other orthogonal ideas that worked better in other fields, we can make a start at arriving. This exposition merely takes what you already know about things and reframes them from the perspective of the knowledge contained; remarkable factors explain how and why they work in the general sense. You will find most of the knowledge I present herein takes existing knowledge from forward of concepts taught many years ago, and relooking at those same things from a deeper knowledge base and fresh perspective to make their deeper, profound knowledge clear. But this simple mindset follows my deeply held principle that tomorrow's discoveries appear from yesterday's assumptions (axioms / assumptions / principles).

The first, obvious one is Dmitri Mendeleev's Periodic Table of Elements. Odd, as the war between Russia and Ukraine flares on today, this book describes a part of the shared culture of Europe and Russia in it's importance as the best example of science in usable form. It was Mendeleev that used his spreadsheet, today's word but an unfamiliar one to his time, based on the atomic sizes and proton counts, the basis of chemical reactions, to predict how newer atoms would behave. Imagine if you had to explain and understand the basic tenets of chemistry without having a universal Rosetta stone for all chemical properties and their implications; how to divide chemicals into families of similar compositions and behaviours. It allows one to create a new element and insert them, reliably into the larger database of chemicals with predictions and rationales. As a work of science knowledge, a cross reference method - placing the elements with the specific information for each and relative information between each, it made a powerful way to solve the problem: and the first problem was realizing how important those variables were in relation. Unmistakable when presented in the proper format.

If I could describe the Periodic Table of Elements in computer terms, it's like having a switch case to choose one of a number of functions to accurately predict the variables and how they react. It had missing elements in the matrix that allowed for prediction based on proton count how a missing element would behave. And true to form, the filled in blanks acted as prediction. In a way, that's what software does: it takes one person's model of what something should add to, look like, compare with, and it places that knowledge in those formats that make it graspable as James Doohan's missing middle finger.

The second useable knowledge format that drove innovation is the old mechanical device plate books from the 18th - 19th Century. As an exposition of simple, proven, mechanical armatures, force transmission, pulleys, belts and other crafted mechanical devices that would predict how a machine installing these elements would behave. Imagine trying to spend your time explaining to people that have very little reading comprehension, and almost as little math, how to understand the math of friction, stiction, three-dimensional moment arms and so on. And, when it comes to efficiency, years in a high school classroom or college sophomore drunken stupors are obviated by a simple craft made from the plates that describe in as much detail as needed where one must cut material and where one must fill material (constructive solid geometry) to create an object like a cog. People that are good at and happy to work with their hands will find a mechanical plate book easier to understand and easier to copy that a textbook full of math. The means of medium is quality from the perspective of the consumer. The number of people that could reproduce mechanical mechanisms[2] that work in real time exploded.

Mechanical drawing plate books were like 2-dimensional math functions that constructed a mechanical object set meant to interlock, and allowed the human to play with them, push and pull and witness how much force they resisted or how little force it took to move across each other. Add oil and felt, one for sliding viscous friction reduction and one to deaden motions and diffuse them, and one allows simple folk to make reliable devices. Two dimensional drawings, a set of tools, materials, oil, and felt is all that's required. It's modifiable (make changes/improvements), reproducible (make many copies), and expandable (extend the range of applications). Think about it, you use wood for simple mechanical actions that don't require large forces, you use soft metals like tin for larger forces, and you use hardened steel for forces in the kilo-N Newton range. The mechanical objects are all scaled duplicates. That makes a tremendous range of mechanical devices in the hands of anyone.

The third exemplar of ideas in a useable form is the Schaum's Mathematical Handbook, mainly written by a small team of mathematicians for the Schaum Publishing House. This book allows people to use math formulas, charts, and mathematical nomenclature that they would otherwise couldn't or wouldn't have the exposure, education, nor tutorship and supervision, to apply the foundational mathematics to derive the simplified mathematical formulae for use in many similar equations. While it's aimed at college students, it allow anyone to substitute the equations and work out the answers for themselves. It's like the Magician's book left in the hands of the Magician's Apprentice. It's accessible

[2] in this rare case, not a redundancy

to all, and by working the equations one can teach himself, herself, themself what the equations do.

I must point out that this comparable wasn't on the list until I had to find some old book and paper citations, but Google's Scholar, a search engine aimed as an academic-, patent-, documentation-, and case law- information aggregator that makes the process of finding someone else's reference citations, going back decades and even centuries quickly, easily is a definite best use of ideas comparable; it is a quantum leap better than the old days where one would have to type them all by hand. I use BiBTeX, and this has to be the easiest I've had it in decades copying over .bib references for my bibliographies. There are thousands of man years sunk into the purpose of accurately and honestly reporting where we stole our ideas from, or in other words, our "Standing on the Shoulders of Giants" to an academic standard. I won't claim my references are the best, but they exist. Just look at some old documents for the difference in the ubiquity and quality of references. When we get to the last few entries written per year for each new book and paper, we as a society have climbed a real mountain of academic reuse and fairness to others. At least in this regard, Google is doing not-evil.

The fifth and final is the most logical one for the acceptance as a work in progress, one that needs no further introduction, one that adapts as new kinds of traffic goes on it and can be relied upon to have more use and faster speeds in the future. It's the internet, a once ARPA (ARPANet) and DARPA project (DARPANet), open sourced for the world to animate. It is both a theory of itself and a working copy for anyone that can observe, take notes, collaborate, implement, design, monetize, and live within. It would be hard to dispute the massive importance of the internet to the world today. While there may be Seven Wonders of the World sitting on the Earth's surface, more people gaze at their phones, consistently.

Every example here proves two basic things: one, that people opt to use information in formats that make it easier to apply, and two, of the many ways that one could present that information there are seldom few that survive the test of time. The ones that do master the duality of an easy need that's needed easily.

This sets two very important, intertwined objectives for software reuse: make it present information in a preferable way, and never lose that preferred way, even if that way changes, to avoid discardment by humanity. Many great ideas litter the dustbin of history. It seems the common thread is ease of use.

Simplicity wins, it always wins

You will see many takes on the problem of complexity and more complex systems, taken on in many variant arguments within this book and why it seems to make things harder. Here's a unique theory by Dr. John F. Sowa. You could search forever amongst the detritus of half-takes and musings. I will point at two.

There was a handwringing argument about why "worse is better" as an explanation of why LISP was doomed; when it comes to software & hardware standards, but this was debunked soundly as a strawman made without straw. Another, better realization was made by mathematician John Sowa in that there seems to be an uncanny correlation between the most sophisticated created standards made by large organizations and the untimely demise of the grand designs by simpler smaller and effective upstarts. Here is Sowa's take:

THE LAW OF STANDARDS

BY JOHN F. SOWA

> *In 1991, while I was participating in some standards projects, I sent an e-mail message to my colleagues, in which I formulated the following hypothesis:*
>
> *Whenever a major organization develops a new system as an official standard for X, the primary result is the widespread adoption of some simpler system as a de facto standard for X.*
>
> *In the original statement, which is reprinted below, I illustrated the hypothesis with four failed attempts to develop widely accepted standards:*
>
> *The PL/I project by IBM and SHARE resulted in Fortran and COBOL becoming the de facto standards for scientific and business computing.*
>
> *The Algol 68 project by IFIPS resulted in Pascal becoming the de facto standard for academic computing.*
>
> *The Ada project by the US DoD resulted in C becoming the de facto standard for system programming.*
>
> *The OS/2 project by IBM and Microsoft resulted in Windows becoming the de facto standard for desktop computing.*
>
> *The failure of these attempts to establish new standards does not mean that all standardization efforts are doomed to failure. On the contrary, many successful standards have been established for computer systems as well as everything from screw threads to grain sizes for wheat. But the overwhelming majority of successful standards are clarifications and revisions of interfaces that have proved to be effective without the*

support of a major standards body. What has consistently failed are the "proactive" attempts to design new systems from scratch that are declared to be standard before anyone has had a chance to implement them, test them, use them, and live with them. Some new systems succeed, but most fail, and even the most successful go through several iterations before the best configuration is found. Such design iterations are best done in small research projects, not in large public committees.

A hypothesis that explains a fixed set of data may be the result of chance. It does not attain the status of a law until it has been confirmed by observations that were not included in the original data. Since 1991, the most notorious illustration of the law of standards was the failure of the seven-level standard for Open Systems Interconnection (OSI), which was being developed by ISO with major support from governments and businesses around the world. The primary result was the triumph of TCP/IP as the de facto standard for computer networks.

One more data point makes a hypothesis more credible, but the most convincing evidence is a successful prediction about the future. In 1995, when Bill Gates laid out his vision of "Windows everywhere," the law of standards convinced me that Linux would replace Windows as the de facto standard for operating systems. Following is some evidence in its favor:

The Linux kernel is a single code base that runs on everything from embedded systems, hand-held computers, and wearable computers to some of the largest supercomputers.

Instead of spreading to all environments, the Windows code base has fragmented into multiple incompatible systems labeled 95, 98, ME, NT, CE, W2K, and XP. Microsoft is dropping support for the older versions, but they plan to produce a new version codenamed Longhorn, which threatens to make all current versions obsolete.

Most graphic applications are being written for platform-independent browsers and languages such as Java and C# that run on Linux, Windows, and most versions of Unix.

The formerly fragmented Unix systems are becoming more compatible by supporting Linux compatibility packages, while Linux has improved its support for the Posix (Unix-based) standards.

Major IT companies that had been promoting their own proprietary operating systems have switched their allegiance to Linux. The premier example is the biggest IT company of all, IBM, which has adopted Linux as the strategic OS for all their hardware. Other companies, such as Oracle, develop all their software on Linux and port the results to other platforms, such as Windows and Unix.

> *Organizations with limited budgets, especially schools and governments, have been moving en masse to Linux, especially in developing countries, such as China and Brazil. This trend is increasing even in Europe, Canada, and the United States.*
>
> *Therefore, the law of standards predicts that the Linux API (Application Programming Interface) will replace the Windows API as the de facto standard for operating systems. That does not mean that Linux itself will immediately replace all other operating systems, but that all major operating systems will become compatible by supporting the same conventions and interfaces as Linux. Microsoft may continue to produce operating systems called Windows, but they already run applications originally written for Linux, and the old Windows API will be used only in legacy applications.*

"But the overwhelming majority of successful standards are clarifications and revisions of interfaces that have proved to be effective without the support of a major standards body."

This is not a recrimination of the idea of making standards. This is not a demonstration of the revolt of the many against the annointed few. This is not a salesmen's urge to sell old software to new customers.

In my view, it is a reinforcement of the general truism that people use useful things, like tools used for custom jobs or mechanisms like the wheel that are hard to improve upon, not because they are approved but because they work better than other options on offer. Wheels work, have worked, will continue to work into the future. To expend a great deal of effort to displace wheels is the kind of uphill climb it should be whether you are creating an overcomplicated standard or a novel rolling design.

This is an important generalization to realize when one sets out with software reuse. On one side, if you find over complicated software then you have the opportunity to return the masses to your improved standard. On the other side, it is a warning of what not to make out of your software reuse ventures.

A Reuse Framework

This chapter gives the management level overview of important things to consider and include for successful reuse operations. The idea of reuse, especially amongst entrenched senior application and software developers requires additional factors to every job. It makes jobs of everyone harder if you haven't accounted for a way to avoid giving the job to same people, whom will appreciate you ever so with these additional demands. If they were overworked before, and remain overworked after, there's little chance for success.

It might seem like another pot of gold for management, but the reasons why reuse movements in the past became fads is exactly why you won't get that extra value without novel thinking, and reoriented resources. You and I aren't inventors of the software reuse ideas, but we can become reuse innovators by heeding the honest realities why it hasn't taken over in the shorter term. As with anything in capitalism, exploiting a commercial advantage, like getting your software made for $0.50 cents on the dollar invested, can only improve your competitive advantages. This book presents an unblinking, unromantic tear down built from other's project ashes.

I come to this knowledge by unique circumstances, as a defence scientist on a budget, I wanted to exploit other's hardware and software as much as possible to accelerate my own work. I've reviewed over 5000 software code source projects: tarballs, zipped directory trees, and so on, including Google, NASA, DARPA, DTIC, NATO, amongst others. No, I'm not going to tell you most of them, because I don't want to create greater enmity if I can help it, I may need some help some time. I've seen it all. The good, the bad, and the embarrassing. If you are reading this, you intend to exploit my sunk costs and suffering. Well played. Long live capitalism!

Here's the software management reality: if you demand software developers meet another requirement set (reuse goals) in addition to the original requirements to accurately plot out the software function, then you will be imposing a need that isn't top of mind for many people, it's a need that appears (to the old way of thinking) as a problem that can be solved later than getting the damned thing running. That will inevitably result in a non-fit structure, and then rushed at the end a restructuring to meet that unimportant (in relative time) "extra" goal. They will rush the important part, and waste the chance.

That's why one of the best recommendations I make later on is to adopt a newer software management project mindset with the introduction of integrated project delivery (IPD) in Chapter 9 alliance where the owner brings in software mercenaries companies that can salvage software on the front end of development to speed your critical path, and reuse mercenary companies that can turn your made code into another profitable product. More on this later.

Existing Methods Analysis

Like one of the managerial overviews I reviewed for this book, the obvious managerial conclusions that people arrive at are like this:

If you want to reuse software, then:

- Identify criteria for evaluating domain analysis approaches
- Survey existing methods for domain analysis and relate to criteria

- Develop description of desirable characteristics of a domain analysis
- Identify critical risks involved issues and assess

This sober advice is a bunch of truisms. It's what everybody tries. How'd that work so far? I am not saying this is irrelevant, but, clearly, domain analysis can't be the answer. I would argue that the risks associated with stopping work to analyze other code ended in most cases as what we all see; developers spent some time, made a few conclusions, perhaps even adopted a couple of things, and then, running out of time, pulled a couple of all-nighters to catch up. The rest of the code floated to the bottom, aboard the Titanic, perhaps.

Let me describe why: advancement from the old way of thinking to the new way of thinking costs time at a minimum, personnel, and reputation value at the maximum. Now you know why it's rarely successful. It isn't easy, cheap, nor free.

Identify Critical Issues

Let's lay out exactly the kinds of risks you should be expecting and prepared for. If you are running a large software organization, or even if you have a critical number of programmers, the impact of reuse on your entreprise must consider the following and what the impact of asking developers to reuse software (Table 2.2).

These risks present in software reuse as an additional requirement set on top of making software will affect all aspects of a project. It will not be without preparation an obstacle to your continued success. No college teaches software reuse, and of the few papers presented on spurts of investment into software reuse, they are few and far between. There is a society in the IEEE that does continue the work, but I doubt you are even aware of it. There are few companies set up to train others. There are few people that have reused software code compared to the vast majority of the mindset of do it yourself.

Don't throw this book away! All these issues can be used to delineate responsibilities amongst a set of workers, a group of competing interests, that accommodate for them into the plan, ideally at the beginning of the planning cycle! Dread nought!

Define Guidelines for Reuse

If you want to make some headway in a less risky, evolutionary way then adopt new guidelines instead of rules and drift into reuse gradually.

Develop guidelines, not rules: rules make resistant people expend effort to avoid rules. Guidelines accept that perhaps not all value will be trapped in

Area	Issues
Development methods	No matter which design methods your individuals/teams use, reuse changes how they settle each component. Novel design methods may introduce delivery slippage.
Development languages	Any grand reuse momentum will require equivalents introduced across all used languages to capture all value. Or it makes rifts in parts assembled from various sources. A risk that may turn into a nightmare.
Development tools	Reuse will require time and resources to reorient the team to a new set of tools that allow the extra activities to support reuse. Or mandate a new set of people with the right skills. This will add delays and risk of project slippage.
Development personnel	Reuse is a skill, it's a top of mind factor in those that understand the new requirements. Is this a demand you can make of every worker up and down the hierarchy? Are you prepared for the headaches and bellyaches?
Domain application	Reuse in terms of a user application is a duality: looking from the software to the application, and looking back from the application to the software. Reuse can encompass both, separately and together. You can't expect people experienced in only one direction to master the opposite on the first try, thus adding risk.
Brand name	If your organization makes better use of software by reuse, then the better quality would help your brand. Reuse isn't second hand, it's better through augmentation of new over old. Can you suffer a temporary brand value loss converting to a new way?
Organization efficiency	Efficient software organizations can suffer shocks and black swan events better than less efficient ones. There's more time and resources for everything else. There's fewer reasons of merit for action than corporate survival. Can you plan in the way to arrive through a reuse struggle? Might it cost you your career if you can't?

Table 2.2: *Software Reuse Critical Issues*

the initial work, but adopting guidelines, over time, morphs people's habits (the instinctual brain over the deterministic brain) lull people into gradually adopting rather than resisting. There's lots of ways to motivate but in the end you want your staff not wasting time, morale, and effort standing in the way of reuse. Let it arrive.

Use the results of your research into the tools to show developers why these new ways might actually speed them up. Develop guidelines for conducting domain analysis and applying results during software development.

One of the best guidelines is to make more, clear additions to software documentation; documentation will contain *proforma* add-ons for the reuse domain, relationship of reuse to overall of domain development, and recommend tools to analyze software automatically to regain project momentum. I cover some free reuse tools in Chapter12.

Principles for Reuse

If you want to pick good ideas from the many software reuse literature, the key principles that seem to predominate much of the surveyed literature are a good starting point (Table 2.4): for the sanity of all concerned, don't attempt to make the design, the code writing, code reviews, and code documentation more onerous, confusing, and distracting from the outcomes you need. Your software must still work, the customer must be satisfied, your reputation must survive intact, and your must still meet resource constraints.

Principle	*Rationale*
Simple	Simple makes it easier to understand; complicated makes it less likely to be understood. Read the multiple authors quoted in this book, clarity and understanding are common themes. Understanding software is an epistemology problem.
Client-based	Aims at the users at the end. They haven't disappeared as the focus, you are merely diverting on the critical path.
Process	Parallel functioning or views of the same objects across rational divisions. Some of the best reuse may come from just keeping pieces separate and communicate between them.
Metrics	Most demand ways to measure both code reuse and the value of reuse. Money is your sword and shield in business.
Relevant	There must be a real need to justify extra work. How can one make this appear as an existential importance?
Object-Oriented	Things within software use object-like abstraction, encapsulation, information hiding, data abstraction, dynamic binding, and inheritance that resolve into domains and ranges that can be made clear. It might even keep developers from quarreling.
Standards	If there are software with standards included, why wouldn't one adopt them? Work should apply towards meeting standards. If they don't have any, they should adopt new ones. It makes your work more sellable.
Documentation	The need for documentation is universal; meanwhile, code documentation practice is lacking today.

Table 2.4: *Software Reuse Guidelines*

Strengths	*Weaknesses*
Many ways to breakdown and process software reuse: documentation, optimization, commonality,organization, and even marketing	*Code is inscrutable when poorly organized, miserably documented, unportable, unintelligible to but a few minds. Care must be taken assessing if this code is worth the effort.*
Code coverage is easier when the basic boundary cases have already been tested. Observing where it isn't tested gives one the difference.	*It takes more work to extract work that isn't in an easily understood format. The harder to understand, the more the work.*
Fresh eyes means fresh perspectives; perhaps you can solve problems in the code faster than making your own from scratch.	*You don't get anything for free. Added costs of reviewing other's code are never zero.*
	Teams not trained for reuse aren't expert at what you expect them to do.

Opportunities	*Threats*
Teams with strengths can offer to supplement the	*Any competitor that gets to the pinnacle of reuse before the others becomes an upstart with a cheaper per unit cost and greater quality rating. Both together can result in a disruptive advantage.*
There will be ways to make your code better for reuse, and in so doing attract others to help your software.	*Wasting time on losing causes, poorly organized code, is an obvious risk to anyone's career.*
Can you leverage different mercenary teams to improve your overall productivity, for a small piece of the prosperity?	*If you don't pick the right code to adapt to, perhaps a competitor steals a march on your product pole position?*
	If you don't adopt reuse as part of your strategy, how long until your competitors do, first?

Table 2.7: *Software Reuse: Strengths, Weaknesses, Opportunities, and Threats*

Software Reuse: SWOT

Software reuse can be thought of in terms of the common business mindest of Strengths, Weaknesses, Opportunities, and Threats(SWOT). We lay them out here in the table form for presentation and discussion with team members (Table 2.7): If you can convince upper management with the strengths, be honest with your employees with the admitted weaknesses, plan to avoid the threats, then you will reap the maximal of all the implied opportunities.

In this chapter, we've laid out solid ways to bring reuse on board, mindful of the pitfalls associated with software development as a contact sport between people. In the next chapter we expand the problem space of reuse in aid of better designing.

Chapter 3

Untrapped Value: The Problem

Accept What Is,
Let Go Of What Was,
And Have Faith In
What Will Be.
—Siddhartha Gautama, The Buddha

Mankind invests vast resources into software, to go along with firmware and hardware in all kinds of systems, to control the technologies that most people take for granted. And in the end, software falls into disrepair - often languishing on old servers or old portable media, in USB thumb drives now, on 3 1/2 inch floppies decades ago, on 5 1/4 inch floppies before that, on 8 inch floppies before that, on magnetic tape spools before that, on punch cards before that, on paper before that, on an abacus addition before that, and on and on as far back perhaps as Hammurabi's code: Ideas set down on solid means that are used by the software that is cutting edge today, but becomes a distant memory tomorrow, as the competitors build and knock down models that meet their needs (mostly) and then get forgotten as new features and better performance.

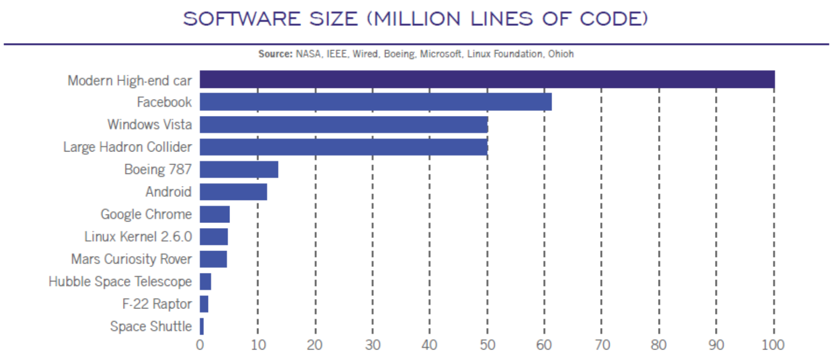
How do we change this? At first glance, it's a problem affecting all aspects of the software model: design, management, documentation, legal, marketing, education, QA testing, project closeout, project commissioning, and maintenance phase.

The most glaring way to improve code is by forcing coders to do less. Automation may hold the key.

Key Automation

There are many things that can be automated to free humans from effort, like:

- Expert knowledge (Make a computer rationalize knowledge)



SOFTWARE SIZE (MILLION LINES OF CODE)

Source: NASA, IEEE, Wired, Boeing, Microsoft, Linux Foundation, Ohioh

Software Code Base Lines of Code estimation, from Autonomous Cars - Part 3: Technology Consolidation. © 2016 Blackberry/QNX

- Domain analysis (machine learn the to and from domain to range)
- Data acquisition (make machines gather and filter)
- Reusable and use standardization storage library (this is my Chapter10)
- Integration with software environments and reuse (development tools that implement reuse automatically)
- Feasibility Issues of automation retrieval interfaces for development tools (ask the tools to find you source)
- Provide consistent, cohesive methods with common interface and understanding (make computers explain the interface to others)

The facts are obvious, there is a growing amount of software out there, just look at the numbers year over year. There's a sizeable code chart that refused to be part of this book but it's good to show data for people to judge by themselves(I turned a lemon into lemonade by making a fair use case out of it).

If you want to know the size and scale of the code bases available, consider this chart- made in 2016 by Blackberry/QNX- of the relative size in hundreds of thousands of lines of code (LOC). Human applications are exploding in lines of code that must work, as software infects everything that once was analog and disparate. From 2016 the source out there is exploded, as more and more programmers bash away at keyboards.

Eschew the One True Way!

This document outlines my overwhelming experience exploring, reading, getting frustrated, getting lost, despairing, getting sober, and accepting defeat from the endless maze of problems around what was some good software code.

The goal of the knowledge herein as outlined provides you with non-specific yet proscriptive practical methods to limit your time wasting along the path to code leverage. This is one of the most valuable books to read, heed, and recommend to anyone plunged into depths of software hell; one that needs to economize on how they approach software left in an unknown state but overcame the knowledge gap and speed towards your already ambitious project milestones, successful testing and evaluation regimes, and infrastructure commissioning and ceremonies.

Take what seems like a one-dimensional cause; in a more holistic way, when one rescues software from an eventual oblivion, you are guiding other people's work into the future prosperity for others to leverage in turn.

Technology, in any age, is like a group of settlers trying to make their way across the unknown territory of the future marketplace. Across the dangerous plains, in the territories of warring tribes, desperate rations, wagon breakdowns, disease, privation, all the doubts and fears seeming mountainous. The draw, the draw of potential gold rushes and open spaces draw those into a journey that will kill many before they reach a path through the Rocky Mountains.

In the United States of America Department of Defense (US DOD), there was a realization of the commonality of software and the desire to economize and reduce redundancy by software re-use. This spawned the CARDS common software repository, and the STARS demonstration.

Where does the mythical distortion of the value in software come from? It was Steve Jobs that famously stood up and complained about how long and expensive software development was. He complained how expensive it had become, all the while colluding with other silicon valley global corporations to hold down wages and remuneration to top performing personnel by not poaching them from one another. Apple, Google and others (not name shaming but proving the point with names people are more likely to know, these weren't alone) underwent many lawsuits from unhappy employees held down by the board room.

And in the rush to make software more economical came the worst language of bloated code and excess, as a direct misunderstanding from those same players as what good code should work like. There came C++. It was hailed as the great white hope for faster and more effective software at a reasonable price. It was lauded and celebrated and adopted all around, new sets of fresh young Ph.D.s launching into a period of Code Renaissance. Soon after, that fanfare went away. People woke up to the reality of C++ code. Remember the dreaded Windows Vista? Did you hear about all the service packs / service patches they had to toss over the wall to disgruntled owners? Remember the GNU BadVista software campaign? Steve Jobs mimics calling it a "Vistaster"? Well, there was C inside as the low level kernel. But the rest was C++, including .NET framework, and C#. While Steve Jobs wanted to improve code with new software ways[1], the people developing OS X for Apple kept the BSD kernel. So even as the corporates wanted to improve costs, what they unleashed was a problematic programming language that didn't handle the underlying garbage collection, the instantiation, and deconstruction of software objects, plus a raft of new DRM digital management, and profile bizarity, as well as expected by group of people unfamiliar with the internals and therefore helpless.

It wasn't just operating systems: TAO - The ACE ORB, MIRO, and many other bloated C++ development systems made larger and bug prone code that was written into a generation of technical papers that concluded about the flaws and failings of bloated software.

After some momentous losses attempting to dethrone Linux as the operating system of choice for servers, most corporate software companies like Apple and Microsoft called a truce with Torvalds.

Whatever your computer interest back in the 2000's, the fundamental need for code to be reliable and cost-effective met the horrible bloating and confusion as C++ libraries caused a raft of problems where otherwise plodding predictable C had done job better. It was a reality check for the value of software done within the bounds of a human understanding always works better than the automated software systems that aren't visible or transparent.

The Linux kernel, through all this, was using the same 1970's Unix software technology better and faster than "new and improved" software development. Are there add-ons within the Linux kernel code base that are C++, probably, but the reason C often beats C++ in performance and bug issues isn't the grammar and the syntax, it's that C is EASIER TO UNDERSTAND, not that it lacks easier to use but hidden features like in C++. This revelation seems bizarre to most, but that's because few people have suffered under the lashes inspecting many code bases, as I have - over 5000, including C, C++, C#, FORTRAN, ASM, JAVA, and others to see WHAT MISTAKES and failings normal programmers make. And my review of the least pretentious and working C++ are the ones that don't use any of the overloaded classes, inheritance and so on. The most predictable C++ is almost all C by features. C defeats not because it's adorned with automated features, not because you can map it out correctly in pretentious Stroustrup. It's not just what software does that programmers

[1] Steve Jobs' original rationale for C++ was to reduce the costs of software development, and in a neophyte karmic redistribution, it ended up in opaque code with odd behaviour as innards of objects change without warning. Forcing additional man hours debugging.

understand, but what it does that software developers DON'T understand that causes problems.

I use this background knowledge as a pretext for the ways and outcomes you wish to arrive at when you reuse code.

The Costs of Bad Software

The cost of bad software is a real problem, and anyone can fall victim to it, literally and figuratively.

From Russia Television:

> A Pentagon review at the beginning of 2020 found more than 800 software flaws with the F-35, and defects that rendered its gun unusable. Though "most" of these issues were fixed by that summer, a number of issues that could "injure or kill pilots or otherwise jeopardize the plane's security" remained. At a current lifetime cost of around $1.7 trillion, it is the most expensive military project ever undertaken, and even with the jet entering service, it already requires costly modernization upgrades. - https://www.rt.com/news/553992-navy-commander-fired-f35/

The costs of software problems isn't just lost deliveries and stolen cryptocurrency, it could be planes falling out of the sky. It doesn't matter how much money you spend nor how many hours you invest into the program. Let me underline the gravity outlined above by pointing out, as an control theory engineer, that the F-35 avionic design is inherently unstable therefore it requires computer control to stay flying. The only way to increase the performance envelope of a manned fighter was to go beyond naturally stable designs. So software is needed to keep the pilot aloft, not just fighting. Pleasant dreams to anyone under an US Air Force base.

The most expensive project in history, the F35 modernization at $1.7 USD trillion, suffers from the same plague of software problems as the smallest app on the Apple iStore. It's a pervasive problem, but will exacerbate situations when the absolute complexity of the software, the larger then the more complex, is hard for any one or any team to understand it all properly. It's as if complexity is a mountain and the higher it gets, then the steeper, wider, and longer is the resulting climb over the top. The cost of bad software is DERIVED FROM COMPLEXITY. Because mistakes are made not just from simple causes but losing the thread of all the ways data changes as it goes. Put another way, the desire to do more things, from more places, unto other places in the software sets up a snowball rolling effect of flow through software, like data flow diagrams that could be printed out on 100 legal size pages in landscape orientation and then laid on the floor from end to end, causes anyone tasked with describing all the places that data goes in a single flow that has to stop and take notes many times along a single finger path because no human brain can keep all the places correct as the list grows. Did you follow that very long, run-on sentence all the way through? It's not 100 pages. Software at the core are ideas, so at the whim of the human mind tasked to change it.

In this next section, you are going to read sentences that have a multitude of complexity layered on top and underneath the words you read. Many things I write are intended to deliver upon many channels at the same time. It's why you will have my words spring from your memory somewhen in the future when your brain makes additional connections of what was writ. That's when you will realize how intelligent I am, and how hard it must be to operate under a brain like this. You may or may not be familiar with all the kinds of theories I will brush over, but I urge that of those meanings sitting in the same words, consider just how a number of complex ideas each with their own dimensionality and rules, can be reduced down by the essential characteristics to a base language of abstract categories. As I often remark, if you want to solve a hard problem, then study an even harder problem. Abstract categories are the hardest topos in existence.

Complexity is both a curse and an opportunity: I resisted getting involved in complexity based estimation software for this book because I knew it's a morass of unending ways to confuse the matter without anything really gained other than some obvious truisms like "complexity is hard". But it is so important to new and improved software, it's an unavoidable topic.

Nonetheless, I've spent many years reading abstract category theory, attempting to round out my algebrae, calculi, and set-heavy education and I stumbled upon graphviz twice for different projects. In some ways, my journey to graph theory, a blunted and stunted version of topos theories, took an indirect path through neural networks and other varied connection theories. I started with the idea that nodes are computers and then swung around theories back from neuron approximations, back to nodes are automata (less complex finite state machines) in cellular automata theory, to Hidden Markov Models(HMM) that model probabilistic systems, to control theory state variables as linear and non-linear systems of derived equations, and then back to nodes are 1-objects, 2-objects, ..., n-objects of a set of morphisms that obey a set of functor mappings that are described by the shape of their forward and backward surjec/injectiveness and whether they create mapping shapes that may be identical (isomorphic up to isomorphism) or some lesser shape, to simple ordered pairs representing destinations and sources of edges and nodes. I am an advocate of the generic use of connection theory, no matter how complex those destinations are, and yet the simplest explanations are the obvious least helpful ways to imagine what a source/destination/node/point/object/cell/neuron might be.

Nonetheless, in order to make sense of the wider kinds of places, or perhaps nouns, that one might arrive at, by means of verbs or actions that transform from/to locations, is to present complexity in the raw basic form of a simple directed or undirected graphs and show you just how hard the interplay of places and actions can become very quickly.

A confession about myself

In order to appear as merely a transporter of facts and opinions, and not a motivated card sharp pushing onto you untold misfortune, I will set the record out for why I grew to detest C++.

My deep belief about object oriented code is not rooted in the programming language C++ itself. I can use it as I might. I find that most people that code in it are just too lazy to learn how to code properly. It's not hard. What these higher level programming languages teach through hiding and opaque sections of function, hidden in the bowels of a libc++ that can change how it operates from one unseen update to the next. The crafter of average code is like an Uber user. An leaser doesn't care about the dents, he won't go and clean it for a fresh ride on Monday. He does what he's told and probably has no more time than that to make it work.

What you want in a software engineer is a programmer with craftsman-like professionalism like the technical guilds of the middle ages. Not the rent-seeking lazy ones that propserity spawned at the end of the age of guilds but the honest humble ones that enjoyed their unchained conditions when noblemen ignored towns business to fight Guelphs versus Ghibellines wars. You want someone that leaves things better than they got them. You want them to hold off on leaving for beer, spend the extra unpaid time, to make sure it can be better. You want them to live inside your code.

And then there's the scientific perspective. I found out when I tried to use my stable and working Kalman filters for localization into a MIRO C++ code base, it crashed. Not my code, the MIRO wrapped around my software. My Kalman filter was built from the ground up, it was tested at every layer. It generated beautiful and smooth trajectories for movement of ground vehicles, even with disturbances and noise from ground travel, and it never lost it's operation to segment faults or corrupted data. Never, I plotted it all out to show. But when it was added and caused the software made or used by others to crash, they had no idea why it didn't work, they couldn't diagnose what was the cause in time to use it for a demonstration. But in reality, it was never my code that was the problem, it was their understanding of a massive software base they didn't understand that was the root of the problem. Imagine if you are the one that can't know where your data was morphed before it go to you, and how long it was delayed when it left your software through a hidden level. It's a big deal if your software has timestamp inaccuracies and that software is designed to run a robot in real time. Looking at the error statistics for a mobile robot, errors and noise in orientation - the angles you predict have changed - are the greatest sources of drift for a computer driven robot by order of magnitudes more than a few centimetres of translation drift. Most people use very little more than templates and class inheritance, some operators that make it look easy in the code but in reality it just leaves people guessing as to what happens with standard object code beneath. It's all capable of being reduced to smaller simpler code. How can I make that grandiose claim? Because, in parallel, the RISC - reduced instruction set chips - from ARM are winners in the great electronic miniaturization of computing hardware application over other chip designs with CISC - Complex Instruction Set Chips CPU's (Computer Processing Unit) and GPU's (Graphics Processing Unit). It's why Soft Bank's CEO bet heavily on ARM for the exploding smart phone / cellphone market of the future. The hardware is an objective layer for comparison and lower power use and faster clock cycles from a RISC architecture in no way prevents one from completing tasks that run on CISC chips with less instructions. This is the local winner of the perpetual hardware arms race. In parallel, one is capable of mimicking or duplicating some of the C++ behaviour with some formalisms on how one writes C software.

I decided never to allow my software to include opaque interfaces that can do anything naughty or unseemly to my data beyond my sight. I would rather make my own reduced capability object-oriented code. And that's what I did. And do. I create object-oriented data structs with pointers to functions to replicate the inheritance of classes. I use file borders to allow for code hiding and data abstraction as a way to reduce security. At the same time, I reduce superfluous functions or methods into #define macros to reduce the code branchings. Anything else I leave to the Gnu compiler collection (GCC) compiler for optimizations as needed. I rarely use operational code, most in scientific prototype code. If you do this, and hide lower level structs into hidden files, you find faster flowing code without the need for bloated garbage.

For a good starter on GCC, there's a refresher by Brian Gough called, An

Introduction to GCC[8] that's a first blush exposé. Of course, if you wish the complete detailed nitty gritty details of GCC there is the book, Using GCC[14] by Richard Stallman and the GCC community coders.

Let me give you a more familiar example. I had a girlfriend living with me for a few months and after some time I agreed to let her son and his girlfriend live with us. These two young kids were down on their luck, but they also had major behavioural attitudes that they can't see prevent them from sorting out their own lives. One day the son came to me and told me that the dishwasher was leaking. He didn't diagnose it, he didn't fix it, and he didn't mop up the water. Welcome to the level of the Uber generation. I didn't figure out the problem for a few wash cycles but I eventually realized the rubber seals were dirty and not forming a water tight seal. I wiped them down, and cleaned the bottom of the water outtake manifold and the problem ceased. It isn't just a code problem, it is a generational commitment problem. All the advances from the 1940's through 2010 came from a generation of Western developers that sought out and made quality. They didn't have an outsource to subcontract to, they were it.

I can't claim most are as above, but you will want to preclude the latter as much as possible while seeking to attract the most of the former.

Now, following a *reductio ad absurdum* viewpoint, then why don't I program entirely in assembly language for surely the best performance sheds even as minimal bloat of C? Well, I am not against it, I have programmed assembly for Intel 8086X, 80x86, Motorola 680XX, Freescale MPC 5XX, series, and so on, for small projects. I believe that the principles (strong axiomatic beliefs) of the Unix philosophy allow for easier portability as well as common object abstractions like files that it makes little sense to program for one CPU type exclusively. In fact, with how fast instruction sets for assembly language change as new processors are created, it makes more sense to store the knowledge inside a longer form inside a language like C. If quantum computers arrive tomorrow, there should be more than the lower level to bridge to.

That doesn't preclude allowing optimizations where necessary, and C has the asm protected word for just that purpose. That option never disappears. But with assembly language it takes orders of magnitude more time to test and debug because it is all down at the smallest granular level. But, if it makes sense for your purposes and you don't care about the limited nature of the work there's no reason not to include some assembly language with other code. It's entirely up to you how you program your application.

Now, with my malignance to one particular software (I did code in C++ at General Dynamics for a spell so I can't be blamed as without personal, viceral knowledge) doesn't mean that I can't or won't use code in a language I don't like. But what I want to make clear to you that a lot of the problems that come from using systems like C++ (problems that were predictable yet nonetheless caused a generation of computers to work poorly) have more to do with lazy or non-deliberate programmers not doing a good job of writing software, from clear program data flows and other proven techniques, for the longer term, and in most cases these same programmers were being rushed to meet ridiculous project milestones that prevent their best most deterministic work from shining through.

This book is a way to show people better, and by adopting and adapting things you Eschew the One True Way! find floating around in the public domain, still accelerating project milestones to ambitious levels can be done without needing to start from scratch. Or chaining people to their desks.

Let's make a rough economical estimate. Suppose that working into an existing project of abandoned code will net you a gain of 40% on your investment of resources. I mention this rule of thumb many times in the book because that's the way it will sink into you (if you end up following my advice and getting 60 to 80% net gain, I am certain to get those numbers to collate). Suppose this is true and remains so over many software project renovations into your own code base. Let's say that the original code base was 10,000 hours - which is roughly 5 person-years. Logically, if that was the effort put into each of ten projects, and you exploit each of them one at a time, year after year, that may reasonably be estimated like compound interest. Then, after the 11 years of code leveraging, you may be in possession of a code base worth 289,254 hours. That's taking one person, working part of a year and leveraging what's been done by many teams roughly equated to five people for a year. Of course, some might be a wunderkind solo programmer. Some may be snippets of a large corporation, perhaps even NASA. If these assumptions hold true, then at $50 per hour you would be in possession of a code base worth roughly $14.4627 million USD. As a means to store, retain, and grow value, there are few activities short of war and vices that cause that kind of value growth.

Let's make a moral argument in favour of the economical ones. There is no greater value from a holistic ecological perspective, to minimize the energy wasted, to minimize computer resources needed, the manufacturing carbon dioxide emissions, and so on. To minimize the collective effort into software development that must be expended, worldwide, is to deliver to the world the best in developed software. There will be an upper bound to what is needed for all applications, all programs, all servers. An upper bound to the various designed kinds of porting, building, developing testing, localizing and rolling out. Before that end, there must be a combining and improving of the overall software on offer into a more significant, diverse, flexible, portable, and understandable than exists today.

The eventual better way won't come in a perfect proscriptive way. That's the problem with today's software reuse: there is no one true way.

Eschew the One True Way!

Chapter 4

Knowledge Reuse by Dialog: 1993 to 2022

Do Not Dwell In The Past,
Do Not Dream Of The Future,
Concentrate The Mind On the Present Moment.
—Siddhartha Gautama, The Buddha

In this Chapter, I reply to the work of an earlier exposition of a fellow technocrat expounding the value of software reuse for the US Department of Defence (DoD) military development community, as a form of dialog. I have copied two large sections, under fair use for analysis, of the thesis of Captain Donald F. Burns III; his master's thesis at The US Naval Postgraduate School, Monterey, California in 1993. I asked and got his approval to use this in an novel and honest format. I asked for permission and sent my draft of replies to the sections for his consideration. I know you might be too young to look back on your careers, but for us seasoned people it's rewarding to find out how one's work was received and appreciated. And appreciation is my best critique of Captain Donald F. Burns III's work I can summarize: his pragmatic grounding in the project development culture, and a thorough review along the software design development of DoD makes his perspective as a project manager looking outward exemplifies how many if not most technical managers would approach the problem, even in today's technology ecosystem.

I would like to use this for two fold reasons: one, that it exemplifies the ideals of reuse are both universal and timeless as a form of demonstrating how valuable reuse can be, and, two, because using his valid ideas from almost 30 years earlier is a great comparison of time and progress, how old problems have and haven't changed from back three decades; it's a reflection of how hard, impractical, and unmanageable the ideals may be. By comparison and contrast alone, we realize even the simple presence of problems today haven't an easy answer if they haven't changed in 30 odd years. It takes at least two things to instill change: a realization that some things haven't changed despite efforts, and an acceptance that to change things one must alter how one goes about achieving that change.

I would like you to accept Captain Burns III's words as perfect and unassailable because it is unfair to judge any work from many decades ago given that as the reply my work has the comfort and conceit of extra time, knowledge, experience, and so on. This chapter is not made to make my work better, but demonstrate how melding ideas over time makes for better ideas by leveraging what was, which is the goal of this book.

The first section is Captain Burns III's Software reuse and the Army program development process Chapter III Section C: OPPORTUNITIES FOR REUSE:

Chapter III Section C: OPPORTUNITIES FOR REUSE

> " The first step in applying reuse to any new system is to examine the operational domain of the system under scrutiny. In essence, if the system to be upgraded or developed is a missile, then other missile systems should be considered as candidates for potential reuse contributions. Other areas which should be targeted for review for reuse application should be systems which incorporate a particular characteristic or functional capability identified as being conceptually necessary or required in the concept exploration phase of program development.
>
> During the System Definition phase, a close examination should be made of existing architectures in similar systems. Reuse requires that these architectures be examined for advantages and opportunities to incorporate both hardware and software technology systems into the new system. Although fielded systems generally contain less state-of-the-art or leading-edge technologies, reuse of existing or architectural concepts preclude large expenditures on concept exploration -- essentially eliminating the re-invention of the wheel. Even something as seemingly minor as the approach used to develop the architecture in an existing system can be utilized to save time and money in system development. Any reuse of architecture will necessitate further investigation of the existing system for reuse of other components.
>
> Reuse of existing systems specifications and associated documentation can provide baseline requirements for new systems as-well-as systems un-

dergoing upgrade. If nothing else, a comprehensive review of requirements for currently fielded similar systems should highlight shortcomings in requirements identification and definition. Test procedures, standards, and results of similar systems should be examined, again with the idea of baselining. As mentioned earlier, testing is critical and can lead to devastating results in the end product when incremental testing shows the system development to be on target, but the final product falls short of overall performance specifications. Only after a project is complete and undergoing operational testing does it become apparent that the system is inadequate, usually requiring more time and money to fix the problems, if they can be fixed at all. Reuse can potentially save considerable time and money if applied at this stage of the software development process. Even if actual requirements and specifications systems are not reused, the contribution of examination and comparison will pay valuable dividends even if demonstrating what not to do in software development.

The next step in the software development process, preliminary design, is the first phase in which actual coded modules can be examined for potential reuse. Applying reuse at this phase requires examination of those systems identified in the previous phases and as having similar requirements, specifications, performance characteristics. Careful scrutiny of selected target systems while simultaneously establishing the preliminary design should provide opportunity for the developer to compare and contrast actual modular breakdown with proposed modules. Because reuse in this phase entails identification and comparison of functions or hierarchical modules which can contain hundreds of lines of code, it is possible that actual code as-well-as module concepts or functions could be utilized in the developing system.

Reusing actual code will substantially reduce time spent on detailed design. It may be possible to incorporate directly (or after slight modification), any reusable modules identified during the last step. Direct injection of reusable modules will eliminate time spent on detailed design during the coding phase. Additionally, this will also have positive impact on related testing, providing early information which could potentially impact other modules and associated testing. If direct reuse of modules or hierarchical functions proves impractical, reuse can still be helpful in the decomposition of the proposed system. Although not absolutely certain, the probability is high that new systems, unless incorporating radically new technology, will have some commonality of function or design with existing fielded systems, thus offering potential for reuse of decomposed forms.

During the coding concept and testing phase of software, the traditional concept reuse is applicable. That is, the traditional concept of line-by-line review of coded software, classified by standard domain analysis. This approach would be used for those modules which have not been the recipients of imported reusable modules identified in previous phases. Although it may be sound easier to commit these detailed modules to standard encoding procedures, the developer still runs the risk of generating errors in the code. And, as mentioned earlier, testing must be considered, developed, and tested. Testing of the new code can be time consuming. And while a line by line search (by domain) can be time consuming, it may still be quicker and cheaper than developing the individual coded lines. Reuse at this point also offers the potential of utilizing previously debugged and tested code, thus saving time. As sufficient code becomes catalogued,it may be someday be possible to draw nearly 75 percent of new programs from reuse libraries, either as functional modules or as individual lines of code, with the remaining 25 percent consisting of the requisite software bindings and new code to utilize technological advancements.

Although integration and system testing cannot directly benefit from reuse, the two areas will benefit indirectly. Because reuse can significantly reduce design and development times and thus the time spent on the associated testing and debugging cycle, the program should have more time in the overall schedule for integration and system testing. Additionally, the use of previously developed, and successfully tested and deployed software systems components can substantially reduce the integration debugging process time.

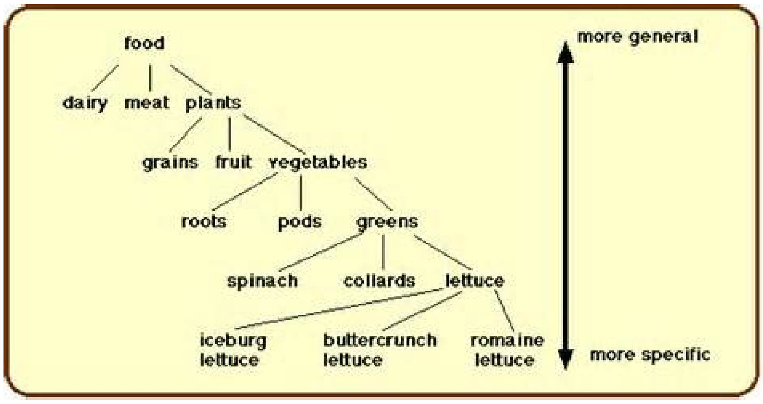
The maintenance phase of reuse candidate software programs should be referred to throughout the development process of new programs. Each step of the candidate programs for reuse should be analyzed for potential inclusion in the developing software program and then cross referenced with the maintenance documentation to determine faults or problems in the original programming. Any anomalies and errors detected and corrected in the original software and its test plan can then be applied or implemented into the new development. If correctly documented, the maintenance phase of programs targeted for reuse provides corrective guidance useful in cutting error detection time found in the original programming, and prevents repetition of errors found in reused software components.

Clearly, software reuse is applicable throughout the development process. It can be both cost and time effective, but there are limitations that should be readily apparent. Only successful, well-documented programs should be candidates for reuse. Programs which suffered from habitual teething problems during development should not be used, even if the program has been fielded. While a fielded program can be judged to be at least marginally successful, software which suffered through slow and error prone development generally suffers from poor planning, design, and execution, thus providing a poor model for new program development. As with any program development, a critical analysis thus should be performed of the risk involved with reuse candidate programs. Obviously, high risk programs based on dubious software programs should only be referred to for the lessons they can provide in error analysis and detection."

A Reply to Captain Burns III's Chapter III Section C: OPPORTUNITIES FOR REUSE

To start, when Burns III talks about operation domain, in terms of military speak, is the application domain of a war fighter. Many Object Oriented Programming (OOP) Layers exist between the application and the foundation code leaves many objects across any bespoke object hierarchy of the operation domain; e.g. missile telemetry needs system plant model for control of motion that is governed by physics laws and therefore a set of physical constants and variables. While constructors are constructors, software objects like Kalman filters, extended Kalman filters, and so on that estimate movement with a probabilistic approach with differing physical mechanics and dynamics, have a set of functions that they use to work properly that are unique for a missile versus a tank. The physics equations are different, but most software Kalman filters are like a toolbox that could be applied over a large variety of applications (operation domains). To describe it in a conceptual way: every software program is like a set of functions as blocks that build a unique pyramid from common bricks. In this perspective, the reuse can be applied most readily to the common parts of the operations design, and less readily to the unique parts. Like Figure 4.1, while a romaine lettuce is a kind of food, it's a more specific kind of food that just "food" that can apply to anything. The more general parts of code are the most common, the least general parts of software are specific to the specific application domain.

From a generic tool box of software creates a pyramid of similar programs using overlapping bits at varying levels that are general and unique parts that aren't. I tried to think of a way to describe it better with a software specific



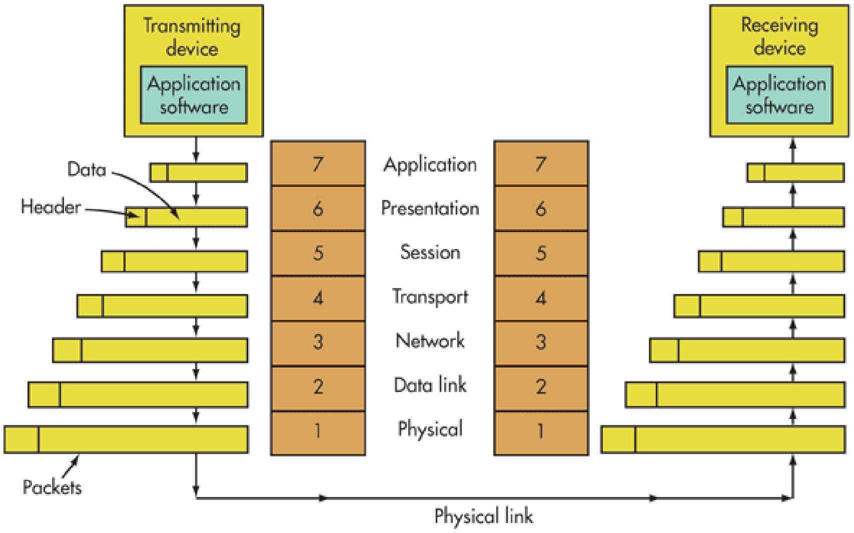
The specific to general, advanced to common is a common truism made by human ontology

context, and then I remembered the old OSI - (Open System Interconnect)[1] layers describing internet traffic from one application on a client talking to a server across the internet. The seven layers are Application (nearest the user), Presentation, Session, Transport, Network, Data Link, and Physical (the wire connections in a wall governed by electrical physics laws) layers. Notice that, in every stage, a packet gains and then loses size as that protocol is added to / stripped off the message and passed onto the next layer. Without considering the actual software flow, this represents how the common pieces of data through transport are larger while the least common (Application layer) are also the least general elements of software.

Looking at the code from another perspective, it's important to recognize that the implementation architecture at many levels can be substituted with something from another kind of process and still execute to similar performance standards, More than identical performance, the same need can be satisfied in many ways. Here are simple examples: you can substitute for TCP/UDP local network protocol from a TCP/IP internet protocol in a communication modules (in fact Linux puts both of these available at the kernel level). From Figure 4.2 above it shows that merely changes one or two layers in the overall process.

If this is the case, then why is it so hard to determine how reusable code can be? It's a reality of the nested system confusion: hierarchies of hierarchies. In any software program, server, library, or core module there are systems within systems. It's not clear, especially when the documentation is lax, just what inter-

[1] https://www.electronicdesign.com/unused/article/21800810/whats-the-difference-between-the-osi-sevenlayer-network-model-and-tcpip



OSI Layers by message size as seen across the seven layers(from OSI 7 Layers); layer headers are inserted and stripped off sequentially on both sides of the data transmission.

acts with what and where does it head: what data types are needed to get the whole thing started and initialized, what data can be saved for later, and what is extra fluff that can be ignored. The ultimate question for every code review: *how can I get what exists to do what I want it to do*?

But how can this be if they spend so much time looking at it? Because complexity isn't easy for most to follow. There are many pieces that can be swapped, but if you do you might have to change the format, decrease the width of the data, renumber outputs, and create new ways to handle an interrupt, and on and on. For example, while the Linux operating system gives you simple function interfaces to provide input and output into the Linux kernel. But, if you don't use the API they give you, there's no guarantee you could duplicate it in an easier way. It will take far longer to find a faster way to duplicate the same behaviour, than just make a test program and send data then read the returned output from the kernel. It's always possible, but it's rarely economical.

What this comment does highlight is that there are identical parts and similar parts, then perhaps there are opposite parts that do something we can predict is the opposite of the first parts. Does any code source give you synonyms and antonyms for functions? Again: different time, same problem.

If one catalogs system components at different complexity/ granularity/scale levels, one can make better use of the available software components. Yet how many software code bases allow easy access to all layers at any granularity? Very few, and hence the problems pointed out in 1993, exist today. Has anyone fundamentally changed how they teach software WITH AN EYE TO reuse? Does every design step have a “research existing software” phase?

There’s another reality, you can find granularity to how things work. There might be a function that works on whole files, while at another level it is reading line by line from the file. There are functions that send data one bit at a time, a byte at time, and others that send internet packets of 1500 characters and compile the returns into a 5 MB file. The programmers have had to work out how to marry the gigantic and the minuscule to accomplish all the tasks expected.

Another complexity difficulty for both , some programs can be dynamically hooked to many libraries that provide data, if you change the configuration the same program changes how it uses different libraries, it breaks older interfaces, and sometimes you can’t realize at the GUI anything has changed. You can even copy over a library with new code and unless it fails you won’t detect it until the library uses the new code. There are system warnings and so on, but remember there are also network-based techniques like remote procedure calls, callback functions, that access system components invisibly to the user and the developer. They can change running systems.

Burns III makes an interesting point about testing inside an application, and identifies direct injecting "reuse modules" that would be needed for testing of specifications and requirements. Direct injection of libraries is a common practice on many operating systems so that’s a If a reuse module is a library of modified / upgraded code components, then it’s vital to be able to test these separately as well as used together inside a program. Do many software libraries provide parallel testing of modules at all levels, all granularities?

It’s important to make a distinction of how software generally worked in 1993, for perspective. When this paper was written in 1993, the conventional nomenclature was a program runs in user space, a server works in remote machines connected by some communication format, and applications were business programs that corporate offices used like word processors and spreadsheets. You may not be aware, but applications in cellphones are actually operating inside the operating system space inside smart phones (back in 1993 a "smart phone" was a personal digital assistant with spotty internet and if you were really posh a cellular) which is different from the 1993 business applications that were whole scale programs loaded into user space on a machine. If this is confusing, I accept I’ve failed at making a distinction, then the way things work today is radically different than the way they worked back then.

The frames of reference for any program designer are vastly different, but on the outside we are oblivious to the radical changes. There are blurred lines around all these different ways to program software, all contributing to the complexity we all take for granted. There were no virtualization platforms that allowed one operating system to work within another program at different scales. There were few GPU's that were as complicated as modern CPU's to offload all the graphics work onto. In 1993, there weren't application specific integrated circuits (ASIC) apart from CPU's that contain the effort built into many old software libraries, only faster and more efficient than software. Today, one can expand the complexity along many dimensions that weren't seen as possible back then.

At the end of any analysis for any application domain, you are left with the basic things that define software: there are hierarchies of objects formed into files that are clearly scoped modules, there are libraries that combine a group of modules pulled together and archived to find objects within them. There are development modules that are being created, they get combined with working code and the outputs are tested to see where they make mistakes / errors or if those errors stay hidden until some other module exposes the failures.

At the design stage, reuse of code will work with no effort spent if the requirements, specifications, and test outcomes involve a pre-made module that meets or exceeds the new design's requirements, specifications, and outcomes. It will require no rework if there are tests available for every granularity, for boundary conditions for all specific to general functions.

If the needs are identical, then no work is needed (development, integration, testing) and the project can move past this module. If the module doesn't work as needed, then a differential analysis determines if and what needs to change to make those requirements meet. If the module can't be salvaged or modified, with an economical plan, then there's a case for starting over from scratch. If the modifications can be made, they are planned into the development plan.

Here's another very serious problem; standards and requirements change. Not only do software conventions force changes to meet new design paradigms, evolved software QA (Quality Assurance) guidelines, accounting standards, contractual arrangements, legal liability issues including strict liability, human-critical and mission-critical systems requirements, Underwriter's Laboratory safety changes, Technical Society (like IEC/ISO or IEEE) Standards, scientific discoveries leading to new hardware, and even now ESG (environmental, social, and governance) compliance mandates to keep investors happy. It's easier today because tools are more powerful, but the mandates programmers must please have grown exponentially at the same time. That means that changing code has become the standard, not the exception.

Looking at a project development process from the macroscopic level, in the era of reuse, the waterfall model may disappear in favour of a perpetual evolutionary spiral process (PESP); it's possible the boundaries of the design process will become skewed to a re-re-re-redesign (*ad infinitum*) of the maintenance software over and over, iteration after iteration. When you tie in the new needs and new problems, it's no wonder why.

We may see an end to the traditional waterfall, the development cycle, and even centralized planning.

Jumping to a comparable, the 29 years of Linux operating system as a project is a resultant of billions of spiral model design processes by many independent or teamed developers, inserted and pulled at random intervals, to then be pulled by (downloaded into source trees with other existing software) others to be tested and a slew of bug tickets complaining about what's broken. And every so often, a team of people stop code insertions and they pull together a release candidate that gets as much of it all working to no one's satisfaction, and then it's released. One day after the release, people are back to inserting more code into the bleeding edge up revision. The entire process is a direct injection code fest, 24/7/365.

Take the concept of a direct injection module, let's call it a sublibrary, testing of a sublibrary on it's own should allow one to send it, complete with it's testing programs, onto another group to verify your results. That different group should find results similar unless there's a particular hardware bug or software architecture disconnect between the two computer machines under inspection. That's all part of wide testing as many have found out the hard way, debugging as they have.

For the sake of like comparison, let's define a sublibrary as the smallest computer machine language code that combines to make all the larger software targets like alleles combine to make your DNA.

But the idea of direct injection of a sublibrary, is the essence of the reuse problem from start to finish: one group designs a software module to solve a problem, then they test it and find it's performance satisfactory and then they ship it to other groups that reproduce the results, or find things to fix, or fail to get it working. The realization here is that any reuse mechanism must never lose sight of the core process begins and ends with work transmitted between groups for the purpose of reusing in various applications, operating systems, CPU/GPU/MCU/ASIC hardware architectures, with a greater variety than was anticipated at the start.

It would be most valuable to allow the receiving team to anticipate how well that sublibrary will work in the new environment, to anticipate and allow their design to anticipate how much of the processing time will be used by all sublibraries working together. It would be invaluable to a designer in the preliminary design state to make predictions about how much work is already done, how much processors have extra capacity, and how much memory will be free. The ability to deliver more than "will this work?" has significant implications for tomorrow's designs. This would achieve the early information needed by designers presaged by Burns III in 1993.

Indeed by considering made sublibraries versus developing new ones will help the search and allow designers to focus on missing or inadequate sublibraries from the catalog or alternatives. Pre-existing tests and pre-existing data sets are also important and encourage reuse by giving future designers finished products to use right away rather than wait for an preliminary design (I don't use alpha and beta testing through out this book to keep the system advancement as wide an interpretation as possible) to be fielded to collect data and begin testing. Every pre-made test on their own gives a developer ready examples to consider and what is lacking that will be needed to meet requirements, or meet code coverage. Reuse is needed by all design stages to allow new designs to take advantage of the prior effort: These are all elements of stored value.

A library by my nomenclature is a collection of sublibraries into a complete set of data creators and manipulators of varying models and descriptions. They are assembled for the use of a program or user, most often far wider than any one developer will use. A catalog of functions and data, like the API of a library is what current developers create. This is the way these objects are offered and documented. But there are many ways to describe the functions and apply the code and it's not always clear to the satisfaction of the next set of developers and the eventual users. There are many ways to describe the purpose of any code, it's up to the user and not the documenter what is needed, and that's rarely obvious to both sides.

While a goal of 75% software reuse is a worthy goal and can be achieved, recent history suggests that the only way to arrive at 75% renews demands that the interface to the code must widen to meet a broader and wider range of potential developers for an even larger set of users. To meet the code base reuse goal, more documentation and better documentation are needed.

It's a paradox, the more code on offer, the more it takes to find within the code what the developer wants if it's not documented better. Another facet of system complexity that isn't scrutable from the outside perspective.

Another innovation to achieve that goal would be make the testing units - and even the debugger - provide more explanation to the developer than just the API.

Any library system that helps catalog all the functions as they are added would also make it easier to reap the rewards of reuse.

In the new era, the maintenance phase takes on a whole new meaning in 2022 versus 1993. Today, electronics are faster and cheaper that 1993 kitchen appliances, with planned obsolescence every two years or so because upgrading and improving one device costs more than simply abandoning it for newer processors and larger memory machines. It has become most industries standard *modus operandi*. That would have been unthinkable to throw out a desktop computer or workstation every 2 years. And yet in many ways today's smartphones are superior to many computer stations of the 1990's.

So, in this climate, the maintenance phase takes on a whole new meaning; it is an eternal background activity. New hardware add-ons are designed into the existing hardware core architecture, specific software to use the new hardware is developed, and the toolbox of hardware and software expands for the next generation of third party software developers to make their applications on top.

The maintenance phase today is the a perpetual swirl of foreground and background activities. Very blurred lines, but nonetheless a steady stream of opportunities to inject into the process. that still exists for any software development.

In this climate, a project designer would be better off knowing what code bases are available, what functionality is already available and for short, sharp projects adopt a very simple strategy: **dovetail** or **piggyback**.

For a piggyback, determine if it's possible to get the existing code to be used to deliver the needed data to the new module without touching any of the underneath architecture. A piggyback combines the two into a serial relationship.

For a dovetail, determine what's missing from the existing code and design a module that will work in parallel with the existing code but also delve into the underneath architecture. Together, the existing code and new code form a bigger interface inserted into the rest of the system. A dovetail combines the two in a parallel relationship.

In a piggyback, testing will concentrate on the new module because it's relatively novel and the previous software will have been tested. It slants the assumption about errors to remain isolated to the new software, while exceptions will pop up. The presumption that bugs won't be in the older code presupposes it was tested for the entirety of possible uses, of which no one can be certain if those parts were never used elsewhere.

In a dovetail, the errors will exist in the new module and may exist in the old module as well. There won't be as much certainty as to the isolation of errors. Both components will need to be tested in parallel.

In any kind of code layering and retesting the same code many ways will find more errors than simple code coverage.

In response to Burn III's predilection to judge software at the project level, teething pains in operation and so on, that, given the complexity and layering of any library, it's hard to write off any code as the location of all the ills and another part the location of all the benefits. Therefore there will be a distribution of code qualities throughout any code base, regardless of the entire results.

The ability to review code and find code quality from any developer and provide the improved code in a software catalog for wider use is as important as the programs and applications in total.

Burns III is entirely correct is that the end user, level and user application there will be much pain and suffering on underperforming software. But, as software code modules advance on down to smaller and smaller modules and smaller portions of poorly working, poorly tested, and poorly documented code, there is a way to make the quality therein live up to the higher standards of the rest.

How this would be achieved can take place in many ways, but no matter which prefers to the designer or developer, which goals are the priority, there will be a better code base in the future for others to reuse and therein delivers the value over and over to the rest of society.

Leverage Strategy #1:

Leverage middle ware and application code on top of the Linux operating system. Or RTEMS, or BSD, and so on. If you piggy back all your code onto a sophisticated code base like a proven kernel, you might achieve both performance specifications, reduce obsolescence, and minimize redundancy as the code base underneath evolves for you. Of course, this strategy would sacrifice the need for larger software teams but require a constant overwatch of the rapid advances taking place inside the kernel project. If one adopts principles like backwards compatibility, upwards compatibility, and so on then one can work in an isolated code bubble with certainty.

Leverage Strategy #2:

Use the NASA re-use standards. It's easy to focus on the narrow area of how you work on your software project, but if you adopt practices that are preexisting as a background activity to your work, you have allowed yourself to exploit the reuse lessons learned by others at their expense. Even if you don't reuse a single line of someone else's code, you will make life easier for the next developers using your code.

Captain Burns III's Chapter III Section D: REUSE INHIBITORS

Captain Burns III's Chapter III Section D: REUSE INHIBITORS

This section is another valuable way of dialog by replying to another technocrat's exposition of the struggles within a large organization, with seemingly limitless funding, but with a many stove-pipe defocus and random time frames. Before you read this section, I want to make a declaration. I wasn't aware of this work when I started this book, I found it about halfway through my book's writing. Long after I wrote the the beginning of this book. You will see the parts I am alluding to later on. If only I had read this in 1994!

> *"Although the software reuse procedure for new program development is fairly straight forward, there are a number of factors which have so far prevented employment of reuse on a widespread basis. These inhibitors cover a wide range of areas, but can be condensed into several primary categories. These categories cover*
>
> *1) standards,*
> *2) training and education,*
> *3) management,*
> *4) lack of centralized and cataloged assets,*
> *and 5) legal and contractual issues [Ref.49 39:pp. 1-8].*
>
> *While the management category would seem to encompass the reuse inhibitors from the program manager's perspective, in actuality, all of the categories bear some impact on reuse implementation. Each category contains inhibitors which actually affect areas outside what could be considered the bounds of that particular category. And within the each category,the separate inhibitors also carry overlapping influence.*
>
> *The following is an analysis of five primary categories of software reuse inhibitors. It will focus on the individual factors within each category and their impact on specific areas, applications, or implementation procedures and techniques of software reuse.*[2]

[2] copied verbatim from Burns III Thesis

1. Standards

a. Lack of Standards

The lack of standards in such areas as hardware and software architectures, and commonly utilized software languages, perpetuated by that the DoD requests and requires and by what the industry and contractors provide, contributes greatly to the difficulty of attempting to implement software reuse. This is especially true of the military attempts at software reuse. Because the military uses civilian contractors to develop most software, then each contractor generally has commercial interests which produce software in a preferred commercially marketable language, there tendency by the contractors to develop DoD programs in the same language of choice. Once the program is developed, it is then usually translated into the DoD preferred Ada computer language. (The term "usually translated" is used because the requirement for DoD to use Ada is fairly recent. A large amount of equipment utilizing a variety of software languages has been fielded prior to the implementation of this requirement, therefore it is impractical to expect this code to ever be compiled into Ada. A factor impacting the compilation of Ada in programs currently under development is the loose enforcement of the regulation. For any number of reasons, the software development contractor may be exempted from delivering a final software product in Ada.) However, because this is a higher order language, each line of which may be the transposition of several separate lines of another code which subcommands in turn can be composed and routines, it does not decompose easily or simply when attempting to examine the code it for potential reuse.

From the hardware perspective, the problem seems somewhat simpler to understand. Although two different programs may be written in the same language and may even have very similar applications or domains, they may be designed to operate in very different hardware architectures. The different hardware systems and their basic approach to program execution may effectively prohibit porting or reuse of other program segments onto targeted hardware. Whereas some architectures rely on hardware solutions to specific problems, others rely on software solutions to address those same problems, consequently, the hardware design will dictate the software developer's approach to the program development.[3]

[3] copied verbatim from Burns III Thesis

b. Lack of standard or common development methodologies

Although the DoD is governed by a host of regulations designed to provide control and structure to the development process, civilian software developers are under no such regulation. Each software development company has its own internal program and guidelines. Consequently, the development process described earlier, which so readily lends itself to a structured building block approach and provides significant documentation so easily adaptable to reuse analysis, is not followed by those outside the DoD. As a result, adequate documentation for potential reuse may not be available. The usual drivers of poor or insufficient documentation is shoddy program design and incoherent structuring of modules, both of which dissuade reuse implementation. These programs often suffer a painful development process and are generally plagued with problems throughout their life cycle, making them unlikely candidates for reuse.[4]

c. Lack of Common Notation for Describing Designs and Requirements

Although seemingly minor, this inhibitor to software reuse complements both the Lack of Standards and the Lack of Standard or Common Software Development Methodologies. Just like spoken languages with their own unique alphabets, software languages have their own symbology and notation. A programmer experienced in one language may only a vague understanding of another, rendering any attempt by the programmer to analyze a program targeted for reuse an exercise in futility. Although most symbology and notation has comparable representation in every language, the cost of translation and transposition can be prohibitively expensive for a project with such a dubious potential payoff.

Another problem occurs when attempting to measure the performance of programs against an accepted standard. The lack of commonly accepted hardware performance standards and software metrics prohibits the developer from effectively comparing the performance of one program or program segment against another. This is a very complex concept, as measuring performance is as measuring is more than just operating against the clock. The

[4] copied verbatim from Burns III Thesis

measure of software performance must include allowances for hardware induced performance, as-well-as broad parameters which define singular computations and program executions. The measure of hardware performance on the other hand must take into account how each specific piece of software goes about the execution of its tasks, and negate those effects to accurately measure performance. The lack of standardized metric to test either hardware or software components prevents any real attempt at even examining the broad base of existing software for potential candidate reuse programs.[5]

d. Lack of Methodology for Extending standards

There are literally hundreds of software development and consulting firms in operation in the U.S. Many of these firms are currently engaged in development or consulting efforts with the DoD, and are often in direct competition with each other. Consequently, there is seldom a free exchange of information between firms on progress in either software hardware development. Additionally, each of these firms measures its progress against its own internally accepted standards, and while some of these standards may be shared or subscribed to by many firms, not all firms agree on all standards, nor are they necessarily the latest standards. Precisely because most of these firms are competitors, any advances in metrics, design, hardware, or development processes, are often kept secret by the developers to gain the most from the advancement, either in monetary or technological terms. It is seldom in the firm's best interest to advance the general knowledge of competitors. Current efforts by the industry to track progress and standards of performance consist of deriving information from trade publications, advertising copy, and reverse engineering efforts. Consequently, there exists within the industry no mechanism or body of regulators (other than the DoD) to decide what are the appropriate standards for the industry, how to ensure compliance with these standards, how and when to upgrade these standards, and finally, how to disseminate this information throughout the industry. As with the lack of standards, the lack of a mechanism to promote and disseminate those standards inhibits reuse by acting as a (negative) force multiplier for an already crippled industry.[6]

[5] copied verbatim from Burns III Thesis

[6] copied verbatim from Burns III Thesis

e. Lack of Standardized Definitions of Reused, Common, Shared Software

> *In an industry bereft of standards, it is not surprising that a common definition for reused or shared software cannot be agreed upon. This is a simple situation of putting the cart before the horse. It would be impractical if not impossible to define such things as reused, common, or shared software without first addressing the industry wide problem of general software standards. Obviously, software reuse is inhibited by the lack of standardized definitions of what constitutes reuse. (If you can't describe it, you can't define it, and if you can't define it, you can't find it, and finally, if you can't find it, you can't use it!) Additionally, the lack of a regulatory body or dissemination mechanism adds to the inherent reuse inhibitors.*[7]

f. Lack of Well-Defined Reuse Methodologies

> *The compliment to the lack of adequate and standardized definitions for reuse or shared software is the lack of any well-defined or standardized reuse methodologies. While the industry cannot settle on standard software development models, it would be totally unrealistic to believe the industry can settle on any standardized models for reuse implementation. Again, any attempt to implement an undefined concept across an entire industry can only meet with failure.*[8]

2. Training and Education

a. Reuse Inhibits Innovation and Reduces Competitive Advantage

Within the industry, software reuse is viewed as a rehashing of old ideas and technology. In an industry where there are literally scores of software producers, innovation is a market discriminator. A key strategy in marketing a software product is to discriminate the program from its competitors. This is usually done by a focusing on some new feature or gimmick. Consequently, reusing previous released software eliminates this potential marketing approach. An additional factor that the software firm must consider is the

[7] copied verbatim from Burns III Thesis

[8] copied verbatim from Burns III Thesis

actual software engineering workforce. If the aforementioned workforce views software reuse as a restraint on creativity or a hindrance to innovation, the company may be hard pressed to maintain experienced and talented software engineers. The perception that a firm cannot engage in software reuse while keeping talented, cutting edge people on the payroll, and thus not produce innovative, cutting-edge competitive software products is a tremendous inhibitor to actual reuse application. This is true of the software firm, whether operating in the commercial market or the DoD contractor arena.

b. Lack of Readily Accessible Information on Reuse

Although the concept of around for years, there is dearth of information readily available on the subject. General industry attitude combined with the lack of standards and methodology has left software reuse in its infancy while the primary focus of the industry promoted evolution of program engineering.

Software reuse information is skewed by rumours falsehoods about the subject. This is the result of the ill informed or misinformed generally interjecting their bias into the available information. It is important to remember that programmers are often paid by the number of lines of code they generate, and consequently find it in their best interest to inhibit something like reuse which they might perceive as a threat to their livelihood.

This overall lack of quality information on reuse has stunted interest in the subject and the proliferation of usable information. The same programmers who may see reuse as a potential financial impingement are also the same programmers that write industry magazines. It is entirely possible that information about reuse could be stifled for the reasons of self interest. Without accurate and adequate information or any mechanisms to spread this information, it is doubtful that reuse will ever be implemented on a wide spread basis. [9]

c. Limited Training for Reuse

For those interested in reuse, whether they are managers or engineers, contractors or the Government, there is little or no training available. As mentioned earlier, there is a dearth of standards within the industry. Consequently, very few companies or organizations are inclined to invest money, time, and resources into training personnel to engage in or manage reuse of software

[9] copied verbatim from Burns III Thesis

components. The lack of training also impacts the amount of information available on reuse. There is a dependency cycle in which information is needed to inaugurate training, but training is needed to develop information, Once the cycle begins, it will be self-sustaining. However, getting the cycle started may be a monumental task. [10]

d. Lack of Knowledge and Training of Data Rights and Licensing Procedures

Although this might be considered as a topic under the Lack of Available Reuse Information category, it is really more of a legal problem than an information problem. Proprietary rights and data rights of published or contracted software are by law the property of the developer (except where contracted developers give up those rights as part of the development project) and can be used only under license from that development firm. For any potential reuser, there must be a license agreement, not just to use the software, but possibly to decompose it, alter it, and finally combine it with other code from similar sources. There are a number of problems with this concept.

First, to submit a program or programs to reuse will require substantial documentation of the software and testing. The producer, in order to protect his interests in this process will need to engage in management of the software which, for an independent producer especially, can take considerable time. Either the producer manages the program himself or the firm establishes an internal mechanism to manage this process. This would require manpower, facilities, equipment, and cash. Obviously, this investment must be weighed against the potential profit to be made through licensing reusable software. An additional problem is potential liability for software problems which may come from a firm's reuse software. In an era of intense litigation against over producer over producer liability it could be catastrophic for the developer if his software caused a major software crash for another developer.

For the potential reuser, the problems are even more complex. The potential reuser must either be ready to spend great amounts of time and money to analyze potential reuse candidates, or he must develop a mechanism to conduct reuse business. Although this sounds relatively simple, the establishment of a reuse management mechanism for the potential reuser would be costly and have a tendency to grow. The essential structure would need to include a manager steeped in software engineering as-well-as software contracting. A bevy

[10] copied verbatim from Burns III Thesis

of software engineers familiar with internal projects requiring reusable components would be required to research and analyze each and every potential reuse candidate program. The section would require both lawyers and contracting personnel to work out the details of agreements which would allow the software engineers to examine other programs as well-as use favorable components. And finally, any effort of this type will require a multitude of administrative people to manage the records and documentation.

A final problem which may plague the potential reuser is the fact that many software forms are here today and gone tomorrow. Although the programs are copyrighted, company may no longer be in business. Any reuser is bound by law to license the software for reuse. This means that the reuser must find an organization or person with proprietary rights over the software. This can be a long and tedious process and may not be worth the time and effort to conduct a search versus just developing the software from scratch. Until the legal fundamentals are worked out and guidelines established and firms are ready to make a concerted effort to implement software reuse, this area will continue to be a problem and will definitely impact reuse implementation. [11]

e. Software Common Practice of Redesign/Redevelop Versus Hardware Incremental Development Practice

Within the automation industry, there are two distinct practices with respect to software and hardware development and improvement. Firms generally approach hardware upgrades or revisions using incremental improvement technique. Their approach to software on the other hand, is one of new program development instead of measured improvement.

Essentially, firms utilize existing hardware platforms and apply focused technological improvements to specific areas of the platform, incrementally improving performance. There are several reasons for this hardware improvement approach. First, the pace of hardware improvement moves only as fast as technological advancement. Although revolutionary improvements do happen within the industry, most of the effort is focused on improving existing technology. Consequently, great technological strides or revolutionary improvements are few and far between. Second, is the matter of economics. All of the firms in the industry are in business to make a profit, and each firm does this in

[11] copied verbatim from Burns III Thesis

a variety of different ways. Firms make money on providing upgrade components to existing equipment -- again, an effort to capitalize on technological advancement. Another method for gleaning a profit commonly utilized by the industry is to offer a wide variety of models utilizing common components or technology similar to practices found in the automobile industry.

Finally, and most controversial, is the practice of utilizing planned technological obsolescence and controlled technological insertion. This method calls for utilizing combinations of components which have a limited or planned life-span with respect to leading-edge technology. This planned obsolescence is coupled with carefully calculated technological insertion. Essentially, a firm will time the improvements as a marketing tool to boost sales where current machines offered for sale have lost their technological edge to the competition. This idea ties in to the concept of maintaining a desired level of market share. If a firm wished to maintain its current level of market share, the firm may hold some improvements in reserve to counter competitive efforts by the competition to increase market share. And finally, incremental improvements are the backbone of the lucrative upgrade market mentioned earlier.

Software development on the other hand is viewed as a cottage industry within the automation field. Because software development is generally less equipment and personnel intensive, is viewed as being easier than hardware development. This view is predicated on the industry's lack of standardization with respect to almost every area of software development. Instead of teams of engineers working together to develop improved hardware, the software environment is populated by individuals, meticulously and painstakingly developing and testing a program on the targeted machine. The industry views software development more as an art form than a science. subject Consequently, software developers are generally subject to less management and control than hardware. This lack of tight control eliminates any incentive within the firm to utilize software reuse. After all, creativity is viewed as an asset in the software field and reuse is held to be in direct contradiction to that idea. And much like artists, software developers reflect values and beliefs, consequently, left to their own devices, few opt to review old programs for usable parts or pieces. Finally, because there are considerably fewer resources required or utilized in development, it is viewed as being considerably easier than hardware development.

It is necessary to make a clarification at this point. Software is commonly released in versions, with each version representing an improvement over previous versions. These improvements are usually nothing more than refined or debugged previous editions of current software. Therefore, these new ver-

sions of the software are essentially not true revisions or design changes of the programming. Eliminating version revisions as true improvements, actual software improvement comes in the form of new programs, offering new capabilities, tools, and quicker program execution.

The two different approaches stem from the manner of development for each of these areas. Hardware development is expensive due to it's nature. Development or improvement of hardware products requires expensive laboratories, equipped with state-of-the-art test, diagnostic, and measurement equipment, capital intensive production facilities, expensive distribution networks, and extensive training for maintenance personnel. Hardware is the product of teams of tightly managed engineer operating in a structured environment. Improvements in hardware are incremental or marginally evolutionary versus revolutionary. Firms have found it in their best interest to tightly manage these assets to appreciate the highest possible return on the dollar.Whereas software improvements seem to reflect flashes of individual brilliance, unencumbered by management or large quantities of equipment. Management is less apt to indulge itself in an area that defies standard organizational structure, management techniques, and time lines. Until the industry institutes software standards, individualism at the expense of reuse will remain the norm.[12]

3. Management

a. Lack of Program Office Incentive to Initiate Reuse

Without exception, software development within the DoD has been focused at the individual program level as opposed to a broad-based focus aimed at multiple reusable applications. Because each Program Manager is tasked with the development of his particular program and will be judged accordingly, there is little interest on the part of the PM to go beyond the program mandate. Even with programs that must be tied together, such as the Army Tactical Command and Control System (ATCCS), which requires the interface of the separate software development programs of the five BFAs, there has been no effort to exploit software reuse, structure standardization, or interface bindings. The PM's area of responsibility, which can include both hardware and software development, really only encompasses a small domain with respect to either of these areas. Consequently, the PM has only a limited ability to influence anything outside of his mandated area of responsibility. Coupled

[12] copied verbatim from Burns III Thesis

> *with this limited ability to influence outside areas, is the political danger to the PM of expanding his control or influence into another PM's domain or program. Although the Army presents itself as an apolitical organization, which is true of the tactical and operational portions of the force, it is not necessarily representative of the acquisition and procurement areas. These areas are structured along the lines of civilian organizations and are involved in similar pursuits. The program development and acquisition field is mostly the domain of the Army's civilian workforce, and very much emulates the politics the civilian industry it mirrors. Domains of influence and spans managerial control within the Acquisition Corps are often jealously guarded, with interlopers being shunted, ostracized, or victims of political paternalism.*[13]

b. Lack of Personal Recognition or Economic Incentive for Developer of Reused Components

> *Putting all the factors together, it is obvious that there is no incentive for the individual developer to either develop or utilize reusable software. The developer is generally paid on a by-line production basis, consequently, to produce reusable code or implement such code would be tantamount to reducing or eliminating one's livelihood. Additionally, software development management is generally pulled "from the ranks," perpetuating the relaxed, almost loose management atmosphere prevalent in most development companies. Because of the relatively unregulated development atmosphere, there is little guidance or direction aimed at reuse employment or development. As mentioned earlier, the lack of industry standards and formal mechanisms to disseminate information or govern rights of reusable software serves again to inhibit the individual utilizing or developing reusable software serves again to inhibit the individual software developer from either utilizing or developing reusable code. Because code and documentation are not readily available without extensive legal negotiations and because there is no royalty mechanism in place to reward the developer for his efforts, there is no incentive to move in this direction, especially for the individual.*[14]

[13] copied verbatim from Burns III Thesis

[14] copied verbatim from Burns III Thesis

c. Lack of Trade-Off Mechanism Between Requirements and Reuse

A requirement is by definition, a need. Software is written to satisfy the need. Any differentiation between the requirement and the performance of the software does not qualify as meeting the need. In order to implement reusable software, the requirements must be reasonably flexible or generic in nature. Unfortunately, however, software requirements are not usually flexible or generic. Consequently, it difficult to find the necessary middle ground to satisfy the demands of both. The problem stems from the development process. The requirements for any project are drawn up early on and are the result of mission need statements generated from the user community. Because these needs are drawn up without regard to software development or software reuse, no compromises or trade-offs are established. Consequently, there is little room for software reuse if the requirements are to be effectively satisfied.

d. Lack of Reuse Cost Models or Metrics

As stated earlier, an industry wide lack of standards in software development, architectures, and metrics has effectively deterred the development of any reasonably reliable cost models for software reuse. This is almost the proverbial chicken and egg situation. In order to determine the cost effectiveness of implementing software reuse, it is necessary to evaluate existing software reuse cost models. However, without effective and widespread utilization reuse, valid cost models cannot be developed. As with any new technology, it often requires investment of a great deal of time, money, and resources to begin the initial venture. Only after relatively large and risky expenditures of capital does a company normally begin to see positive returns on investment. The current state of the software reuse industry is much the same way. The current measure of risk has so far inhibited reuse and the associated models which could someday prove its profitability.[15]

[15] copied verbatim from Burns III Thesis

e. Limited Vision or Leadership for Reuse

As stated earlier, management of software development is very much an inside job, consequently, those boosted to leadership or management positions are often interested in maintaining the status quo versus implementation of new cutting edge ideas. This is not necessarily because those elevated to management are against new ideas, but more because of the reputation established by the company before the new leadership took over. Essentially, some companies are known for certain software traits, designs, or architectures which are accepted and expected within the industry. To break with this established convention can be costly in terms of lost customers. However, the greatest inhibition comes general lack of knowledge about reuse in general. For the manager, the requisite questions of how to classify software, where to find the necessary reusable software candidates, how to navigate the legal obstacles to reuse, and how to motivate the actual developers to implement reuse are insurmountable simply because of the relative infancy of the field and the lack of managerial experience or established guidelines in this area. There is one other simple, yet looming reason for the lack of reuse implementation by management. A great number of those rising to managerial positions in the software development field do so by default. Most lack the drive and aspirations found with managers in other areas of business. Consequently, there is a marked reluctance to aggressively pursue such controversial and dubious endeavors as software reuse.[16]

f. Lack of Knowledge and Training on Data Rights and Licensing Procedures

Software, like nearly every other product on the commercial market is surrounded and supported by a host of laws designed to protect the producer's product, his ideas, or development process from being copied without permission or monetary compensation. Copyright laws similar to those covering audio and video tapes and discs govern the software industry. However, unlike audio and video tapes, software, although relatively easy to copy or pirate, is of little of no use without documentation, and of no use unless it can be decomposed. Therefore the problem here is not piracy, but licensing - the authorized use of all or a portion of a piece of software (to include documentation) by another company in exchange for monetary compensation. This is a

[16] copied verbatim from Burns III Thesis

relatively new field with respect to software, and lacks precedents for establishment of guidelines or rules. Because of the vagaries of software development and business, questions and concerns addressing the potential profitability of a program which contains reused software, and potential liability of the owner of reused code, and potential licensing of software components which are composed of new code as well as reused code pose special problems for software development companies. Although many companies are faced with similar problems which impact the decision making process, very few face the situation wherein their product can have an indeterminate effect when imported into another program. The potential outcome of such a situation could be devastating if for some reason the reuse software creates problems or systems failure. The issue of who is responsible, the importer or the original developer, is very much in question in such cases and has yet to be determined in potential licensing agreements. The issue of product liability in cases of failure is of only one problem however. Just as important, at least to those individuals who develop software is the issue of by-line payments. Should the individual developer be paid for the reuse of his product, and if so, how much, and how should this issue be approached for a software program which contains reused candidate for software reuse? This may be moot if the target software is altered to facilitate reuse. It does however, raise another issue - that of technical propriety. If a piece of reuse software is altered to facilitate reuse, does the original developer remain liable for problems? Who controls licensing of altered reusable code? Finally, when the Government engages a contractor in software development, the Government generally requires the proprietary rights with the software. This presents problems when the software contains reused code. What are the legal rights and obligations of the original developer of the code, and what are the rights and obligations of the second developer or reuser with respect to the different parts of the code? Who is responsible for the performance of the code, and who takes responsibility for failure? These and other issues have yet to be addressed by either the Government or the software industry. Until these questions are answered however, potential legal obstacles will continue to be major obstacles in implementing software reuse.[17]

[17] copied verbatim from Burns III Thesis

g. Contractors Not Paid for Productivity

It should be clear by now that the bulk of inhibitors work in concert to prevent the widespread implementation of software reuse. But regardless of the wide range of reasons for lack of implementation, the bottom line in most cases is money. Today, contractors are not required by the Government to utilize reuse. And because of all the reasons stated above as-well-as the implications for the industry -- widespread implementation of reuse would substantially reduce the number of competitors in the business -- there has not been a rush to reusable software. Further, it would be ludicrous to assume that any developer would either design a program to be reusable or utilize reusable code without some sort of incentive, either in terms of more follow-up contracts or direct monetary compensation, while at the same time facing possible elimination from the industry by the very thing under development. A developer interested in perpetuating his business realizes that reuse could be a serious threat to continued operation. As pointed out earlier, developers view reuse with some skepticism, realizing that widespread implementation could change the his to software face of the industry, eliminating some of the players and the way business is done. For any concept with such a potentially catastrophic impact, both the short and long term monetary rewards must be enormous. The current system falls far short of offering this type of reward.[18]

h. Other Potential Reasons

Finally, there are a number of obvious inhibitors that should be mentioned. These inhibitors need little or no explanation, and are listed as follows:

(1) Budget and schedule pressure.
(2) Increased organizational interdependence due to reuse.
(3) Redesign versus redevelop mentality of program offices.
(4) Profit and greed on the part of the developer.
(5) Software is not viewed as an asset in program development.[19]

[18] copied verbatim from Burns III Thesis
[19] copied verbatim from Burns III Thesis

4. Lack of Centralized Catalog of Assets

a. Too Few Libraries

The current approach to utilizing software reuse is to catalog lines of code and provide access to that code through a system of libraries. These libraries would categorize code by a multitude of characteristics and provide the code and its search relevant documentation through an automated and retrieval system. Currently however, there are only a few woefully small libraries currently in existence. Although the library concept is the most reasonable approach to real world implementation of software reuse, it also presents a number of unique problems. First, who is the proprietor of the library system -- the Government, the commercial business concerns, or some non-profit organization? Second, what categories are logical and reasonable for the classification of the very broad spectrum of software? Next, with such a prolific amount of software already in existence and more being produced every day, how many of these facilities will be enough to meet requirements? And finally, who should pay for the initial capital investment necessary to establish a series of libraries and their requisite automation links? These issues will be some of the first and most important to be addressed once the Government and industry begin to accept and develop software reuse on a widespread basis.[20]

b. No Easy Way to Search For or Retrieve Components

Touched on earlier, reusable code will most likely be made accessible through a series of libraries. Although this should make the job of locating large quantities of reusable software components easier, the task of finding the right components within this large reservoir of software can be monumental. A major inhibition to reuse is the time required to locate potentially reusable software that conforms to the needed template. Even with automated libraries, the shear volume of potentially reusable code could literally take weeks or months to comb. It can take twice that much time to test. Another problem is how to search for the necessary code. There are millions of potential categories or software classification, ranging from overall program classification to subroutine and individual code segment classification. Most of these components fall within the domain of several of these potential categories, making the task

[20] copied verbatim from Burns III Thesis

of assigning code to a specific category even more difficult. Any programmer seeking to utilize reusable code will need to have a detailed description of the type of code required. The process could be equated to trying to find one specific piece of an unassembled jigsaw puzzle. Until a coherent and easily utilized search and retrieval system can be established and implemented, software reuse cannot be a viable development tool.[21]

c. Defined Way to Test or Accept Components

Having examined the classification and storage problems as-well-as the search and retrieval, the next logical step is to examine testing and acceptance criteria. Once a developer has waded through the system to locate reuse candidate software, it must be tested to ensure the need or requirement. Currently however, the only test available is to actually integrate the reusable code into the program and test for functionality. The testing process can be expensive and time consuming. Of great concern to any developer is the quality of potentially reusable software. As with any endeavor, there is a right way and a wrong way to do things. The same is true for software. Shoddy or unproven techniques used to develop software make for an equally shoddy or unreliable product. Poor quality software can make integration difficult or impossible. It can also lead to difficulty or impossible. It can also lead to difficulties when trying to layer or host application software on top of an operating system utilizing reused software, or vice-versa. But of even greater concern are the proprietary rights issues described earlier. Should the software prove unusable, is the developer still obligated to pay user and licensing fees? Further, does the library as serve licensing agent or should each interested party engage in negotiation to arrive at some mutually agreeable arrangement? And in reference to the quality issue what is the developer's recourse after discovering he has inadvertently reused a poor quality piece of software? These and other questions must be answered through the legal system, library system, and evolving efforts to standardize the industry before they can stifle software reuse in it's infancy.[22]

[21] copied verbatim from Burns III Thesis

[22] copied verbatim from Burns III Thesis

d. Configuration Management Across the Library Network

Software products generally go through an upgrade/update process that works to create newer and more refined versions of a particular program. Often these changes or upgrades are so prolific and frequent that five or more versions of the software can be in use simultaneously. Often these changes or upgrades are so prolific. The latest version is usually the most refined and error free of all the configurations available. Keeping up with these changes and associated documentation means replacing current versions contained in the libraries with the updated versions. This can be a costly and time consuming process, and leads to more questions. Specifically, should upgraded versions of software be provided to developers currently using earlier versions of the software? Are amended licensing agreements necessary? Clearly, as with the other categories of inhibitors, a great deal of work will be necessary to overcome both the obvious and subtle problems in an industry wide basis.[23]

5. Legal and Contractual Issues

Most of the legal and contractual issues concerning the implementation of software reuse have been mentioned above. There are complex issues with far ranging implication. Addressing these problems through the legal system will take years, and in some cases will result in dubious outcomes. The complexity, cost, and extent of these issues, possibly more than any others, will prevent the software industry from accepting and adopting reuse. Few companies today can stand the withering legal assault that marks a product liability suit, nor can most companies accept the staggering fines imposed for copyright infringement or piracy of proprietary data. Neither do most companies believe in their best interest (profitable) to be obligated to maintain possibly legally mandated technological upgrades of software in libraries, insure reusers receive upgraded software and appropriate documentation, or maintain large staffs of legal personnel to look after the company's best interest with respect to reuse. Therefore, most companies have so far given reuse a wide berth. And, because expense to address of these issues, firms will maintain that distance for the foreseeable future. "[24]

[23] copied verbatim from Burns III Thesis
[24]

A Reply to Captain Burns III's Chapter III Section D: REUSE INHIBITORS

This aforementioned passage is a wealth of information, from legal, to project, to user, to management, to individual, to creator, to software developer, to hardware developer, to education, to licensors, to liability lawyers, and so on. There are a number of grim realities and prescient conditions contained within this assessment. Here are some replies:

Lack of Standards

There are many standards for software by design itself. There are a few proposed software reuse standards and those have existed for a few years. I cover some of them very briefly (because each one is a paper or a book on their own) in Section 5: Proposed Methods.

The US DoD has many standards that apply to reuse, and the most important I've found is the The Department of Defence Architecture Framework (DoDAF) (https://dodcio.defense.gov/Library/DoD-Architecture-Framework/). It has a metamodel (a very important construct recommended by many reuse models) and expands on the 4+1 viewpoints for systems design. Here's the DoDAF synopsis in it's own description:

> The DoDAF enables architectural content that is "Fit-for-Purpose" as an architectural description consistent with specific project or mission objectives. Because the techniques of architectural description can be applied at myriad levels of an enterprise, the purpose or use of an architectural description at each level will be different in content, structure, and level of detail. Tailoring the architectural description development to address specific, well-articulated, and understood purposes, will help ensure the necessary data is collected at the appropriate level of detail to support specific decisions or objectives.[25]

Since 1993, the DoD has achieved a break through in both policy and execution at adopting a common, standard model from which all components (intercontinental ballistic missiles (ICBM), tanks, ships, ...) fit into it. Whether or not the complete DoD entreprise, the world's largest budget, has adopted and follows this model remains to be seen.

The DoDAF contains the following model views:

- All Viewpoint (AV)

[25] https://dodcio.defense.gov/Library/DoD-Architecture-Framework/dodaf20_background/

- Capability Viewpoint (CV)
- Data and Information Viewpoint (DIV)
- Operational Viewpoint (OV)
- Project Viewpoint (PV)
- Services Viewpoint (SvcV)
- Standard Viewpoint (StdV)
- Systems Viewpoint (SV)

ADA as the common programming language has been around since 1993, and has remained as such. That's another standard achievement.

Lack of Common Development Methods

From a very short search of the internet for software reuse on one of many search engines, one can find:

- Protocols API (Communications/File Format/...)
- Standards (Technical/ESG/Safety)
- Guidelines (CMMI SW Levels)
- Design Requirements
- Design Specifications
- Project Lessons Learned
- Repositories
- Open source libraries
- Tools

All of these are readily available with many documents and examples of many, all one search away. There were dramatic efforts in the late 1990's and then again in the 2000's. There's a lot more that's been written, good and bad, about the ideas that make it software for all. You can find many examples of any of these to educate you along many other dimensions of software reuse knowledge.

Another way to fulfill the goals of reuse is to consider all the surrounding technologies. Reuse can take many forms. Computers are, after all information passing systems. The fit, function, format, form, and rationale inside any computing technology revolves around other theories, and very rarely stays static for long. So even if reuse development methods do exist, it's unwise to fix your work onto any time-dependant single technique or technology. Everything becomes obsolete. Perhaps the most glaring example is the waterfall design model itself, in the presence of accelerated development cycle that takes weeks in what used to take years, has become outdated. It's very rare for anyone to begin with a blank canvas, unless they are creating something as unique as it is expensive to design, like quantum computers.

Lack of Common Notation for describing Designs and Requirements.

Since 1993, there have emerged many requirements contexts - ways to define and describe a system design - that meet many different industrial needs. Realtime systems evolved out of the generic computing beginnings. More importantly, from a military / army and perspective, mission critical and safety critical systems designs spawned out of the original flight systems, rocket systems, and other dangerous application requirements. Requirements of many kinds have grown out of the original limited nature of computing as a building-filling machine that merely collects records. This explosion of needs looks in no way to be slowing down, and probably has accelerated.

Nonetheless, one problem Burns III notes that isn't in dispute is the generally underwhelming state of software documentation haven't surpassed the needs of modern users of the software left by older people. Indeed, it's a self-preservation behaviour in some industries, to make code as obfuscated as one can to avoid the older, more expensive software developers to be supplanted by younger, less expensive new developers. Making layoffs harder is an eventuality of many application areas with or without great competitive pressure. If you fire the one guy that understands the hard bits it makes it difficult to achieve even the original success.

Design Standard for many fields exist. In the open there's the 4 + 1 architecture, the DoDAF 2010 Version 1 and 2 Architecture Framework, NASA Study of Systems Engineering, amongst a field of many.

The fact is there are so many operating across many disciplines that have captured, synthesized, and presented for free as the culmination of a lifetime's of systems design experience, working systems, with both successful and disastrous outcomes. A quick search search of the internet brought me thousands of pages of software design, guidelines, and standards. Too much information for anyone to dutifully adopt in another lifetime.

This common notation complaint is a valid one, still. It's unlikely that many will decide to make all notation common; it does seem an improbable likelihood.

Now, on the other hand, many strategies, many use cases, many views, and so on, that can describe a function or process within the finished code. But, and this is the singular realization, does that the code module reflects its position within the original design or the final design. Does anyone close the loop before between the written design and the final code?

Performance metrics are another area for concern as above the numbers and measurements abound. These notations are unlikely to be common. But the real question is does the data go back into the software design and reuse? Wouldn't the value of performance metrics be most useful at the preliminary stage of another design cycle? Coders fixate on delivering into the final working product, is there an opposite stream of effort reporting back for the next design? It seems the most value is stored at the beginning of every cycle, and not as an aside.

If you recall the common infographics about the timing of changes and their relative cost during the development process you will see that changes at the beginning are the cheapest, while those attempted at the end are the costliest. It's a truism that information at the beginning has the most value, ergo.

One difference between the hardware design community and the software design community comes about by how well they appreciate these realities.

If you design electronics badly, if you don't make certain that your design choices are correct, then when you build a test board you find out that either some of it won't work or none of it works through short, dead pins, race conditions, and so on. It's junk. Given this reality, hardware is motivated to make the changes before disaster.

On the other hand, with software, as systems get bigger with more complexity, you can get some of it or most of it working barely even if it's poorly, slowly, or not up to the performance metrics.

I suppose the dichotomy of hardware versus software makes one team almost religiously pedantic about confirming the new designs will work. And on the other hand, another tends to be lax about performance at the start, supposing they will burn the midnight oil towards the end in any case.

Fighting cultural norms is as important as any other factor restricting software efficiency and reuse through better practices adoption.

Extended standards has become another standard within software and within design.

Reuse Readiness Levels (RRL)

NASA has made a pivotal first attempt at standardizing reuse levels. They have created Reuse Readiness Levels[11] (RRL) as means to measure how "ready" they are.

From NASA's report[11]:

> Through extensive discussions, the WG identified the nine topic areas that were deemed important for measuring the reuse maturity of software. Alphabetically, they are:
>
> - Documentation
> - Extensibility
> - Intellectual Property Issues
> - Modularity
> - Packaging
> - Portability
> - Standards Compliance
> - Support
> - Verification and Testing

This is the widest and most reasonable way to estimate all value that is included. If an entire criteria is missing then the value for that subelement is essentially zero. It makes it pretty clear when considering source code from a pure quantitative perspective. However, my experience tells me sometimes the best code comes from sources with the least spit and polish. And support is a relative factor, if someone saved code 10 years ago and hasn't opened it since, it's unrealistic to ask for support. Digging into the code makes a qualitative determination.

Training and Education

Training and Education reuse inhibits innovation and reduces competitive advantage. It is a tangible deficiency that today's coders still aren't taught from a perspective of salvage over fresh design. But perhaps the fact that it comes later with experience makes it a better topic for advanced coders, if they get time and help (AHEM) to make that fresh start.

The Axe analogy: Common - Leading Edge - Bleeding Edge

The bleeding edge of high technology: software made for novel hardware, novel algorithms (at one time octal trees were novel to the crowd familiar with binary trees), and lesser known methods is subject to a rule of thumb: The Axe Wedge analogy.

Here is my analog for effort versus novelty in software: The Axe Wedge (Figure 4.3). The main concepts at play are three: bleeding edge are the novel addons to code bases, normally from inventions on their way to production quality code. Bleeding edge software is usually known by very few people, primarily the inventors. Inventors aren't guaranteed to be the best coders, they are normally fulfilling the hardware working from a naïve perspective. And they don't necessarily care about the finished product (inventors would rather go on to the next big thing than stick around to debug that last invention, generally). The leading edge are elements of software better tested, better designed (perhaps even redesigned) and are less prone to obvious errors. Common are software elements that everybody understands, or has access to. There's little to be done at this level, save understand how to apply practised software.

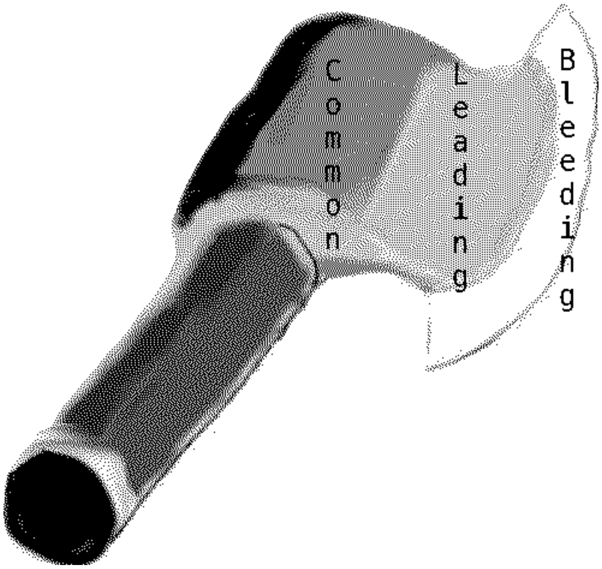
So the general relationship is the more common the idea, the more quality the software should be. The less common, the fewer people that have solved the problem and designed the software version, then the more likely the code is to be of low quality. As time advances, the bleeding edge moves towards the leading edge, and thence the common part of the axe head.

As you examine your next salvage effort, you would do better to assess just how far along from bleeding edge the software is likely to be. With this rough rule of thumb, you can plan adequate resources appropriately.

This is a common and interesting mythical belief. People often mistake the bleeding edge software as the only point that matters when making the next killer app. While making the cutting edge will make the path harder, there are fewer or none that have worked on this particular problem, or very few have succeeded in getting this working.

Invariably, though, the outcome never just rests on that novel algorithm by itself. The rest of the background, the leading edge, the middleware , and the common core must also work, and in physical mobile systems like autonomous cars, that also requires everything else must be working.

Let me give you a real example. In 2005, a team from Carnegie Mellon University (CMU) had developed a very novel autonomous Humvee vehicle with a special perception module on the roof for the DARPA Grand Challenge competing autonomous vehicles held in the California desert. The CMU vehicle, H1ghlander, suffered some kind of failure that prevented it from winning the contest. When it was working, it was clearly ahead but once the fault struck they lagged behind. After ten years, they discovered what the cause of the fail-



The Axe Wedge of effort and novelty: from the bleeding edge, through leading edge, onto the common components of software; Bleeding edge has the least experience, most bugs. Leading edge is better tested, better made but not foolproof. Common are developed to levels we expect.

ure was[26]. It had nothing to do with the novel hardware or software. There was an accident before the final race, and in that accident the vehicle rolled over during a trial run(Figure 4.4). In that event, a broken filter inside the vehicle failed during the race. The more novel CMU vehicle lost to a less sophisticated Stanford's Stanley vehicle. Every component inside a design, whether or not you reuse a vehicle as the base of your design or not, is involved in your ultimate success or failure.

There is a wealth reuse accessible information but factors like corporate uptake or buy-in will force dedicated employees to learn in their own time, in already few hours they have.

Training courses for reuse are indeed few and far between. This is another reason for this book.

Licensing and data rights, including fair use, are in short supply.

[26] https://spectrum.ieee.org/cmu-solves-12-year-old-darpa-grand-challenge-mystery#toggle-gdpr

Left: Red Whittaker and the broken filter from CMU's H1ghlander. Right: CMU's H1ghlander crash before the big race, the DARPA Grand Challenge.

https://spectrum.ieee.org/cmu-solves-12-year-old-darpa-grand-challenge-mystery#toggle-gdpr

The one realization for this is perhaps to contain a great deal of this surrounding material into the repository that supports reuse. That may be the means to break the *status quo*. If one includes the resources close to the developer may end this part of the problem spectrum.

Software Practices vs Hardware Practices

Evolution versus revolution. Is there a way to get the benefits of one without the liabilities of the other? The hardware practices must be aimed at evolution because risking product failure by extending beyond the proper design, test, and evaluate before the product launch makes hardware flow slowly. It is a reality that people will overextend their work in software because the marketplace offers returns. You can't avoid your competition spending on cheaper software monkeys worldwide to advance features that you can't match. However, if you exploit the magic of reuse, making reuse easier, you can make your effort pay off better in the long run. Remember, the more code you write, the more you have to debug. If every layer you make is better designed (model is the number one word from surveyed studies) then you can meet the features race.

Lack of Program Management Incentives to Reuse

This is a priorities reassessment and not a skills, methods, and motivations challenge. The fact is that if people want change they have to pay for it. You should pay for what you want to see in the marketplace, and government, as the biggest purchaser in the national market, must be the leader. There must be the will, and government prioritization to support the value of reuse. Reusing actual code will reduce the time spent on detailed design only when customers are rewarding reuse with money. It should become a requirement to declare how much code will be reused on the new project, and perhaps any donated code by large corporations received tax write offs to produce this code in the open domain for others without encumbrance.

Looking at the motivation from a managerial level, perhaps the same management should widen the user base to beyond the non software users. Many other sciences could use the basic code if there was a better interface, like visual programming plug and play GUI mechanisms. Perhaps bringing onboard software neophytes like college undergrads and high school students will also expand the spectrum. Any web browser is a potential user interface to your software.

There are some developments in current systems, some software reports bugs and error reports right back to error collecting servers for analysis for maintenance development teams. There's a lot of beta testing in gaming software by free time donated by users.

There is one great distinguishing problem between military procurement versus civilian procurement is the problem of scale: the scale for economies. When you design to build 2000 units, there's a smaller profit limit and even larger downside risk for cost overruns and expanding scope of work. If one can create the right motivation to exploit the majority effort for even the smallest unit numbers, to find ways to profit via reuse, then you can motivate every project management. There's even value to be extracted from reuse after any project by second and third parties outside the original scope. If there's a way to get paid for the code by many smaller parties, especially with parts developed by private institutions with public funds, there's a way to coax reuse.

Here's a perfect example of novel project management reuse: in Burns III view, projects with teething pains are not the best for reuse. Taking a holistic view, a management view to review all code, to salvage the best of every project, across the success spectrum, would be a way to educate managers in the particulars of what went right and what when wrong as well as the technical value salvaged. It's a perfect starting place for new managers.

On the government side, projects during the funding process become stovepiped (working in isolation to limited objectives and outcomes) due to the need to focus on the project's limited resources, especially timeline. Due to the process restrictions. expanding mandates and wider goals are not in the purview. These realities force them not to take on side issues, nor delay to deliver group outcomes for other projects. It forces extras that all might provide benefit off the table or to the right on delivery. Work silos are the foundational means that propagate the isolated nature, and limited coordinated projects. There are project offices that begin to unify work, especially the special projects offices, but there's no way all projects will coordinate fully until there's mandated law. Until the mentality changes, projects that go over budget either can't deliver the numbers they plan on or must await further authorized expenditure.

In parallel and largely along the same vein, the civilians in DoD aren't as apolitical as the mandates require. As a way to bypass that problem, if the reuse libraries and search repositories were used at the other side within the contractors, then the reuse would be adopted multilaterally and the impact on the end users will be delivered. There are many ways to maintain a discrete distance of the reuse from any other software-related problem.

Lack of Personal Recognition for Reuse Components.

A valid reason behind project management motivation must be seen in parallel with the motivation of workers: a royalty scheme for reuse by those that refurbish code they reviewed would create positive reinforcement for these impacts. It makes perfect sense to pay for the benefits you want. If an experienced reviewer measures the suitability and the potential value for reuse. Sort of like a salvage yard wander or the MASH field hospital triage. A group acting as a clearance house for improving software along every dimension: documentation, testing, code coverage, software classification, optimizations, portability for other hardware will provide starting points for many others.

c. Lack of Cost models

Cost models go hand in hand with motivation into reuse of code. There are many ways to use measurement:

- Pay for reuse salvage;
- Estimate reused code as leveraged value;
- Pay a share to code refurbishment teams; and
- Keep public open repositories for the public benefit and claim a tax credit.

Any of these may solve the chicken / egg time causality problem.

Limited vision leadership for Reuse

On a whim, I searched my twitter for a twitter handle for software reuse. There was a software reuse account. It was started in 2008. And it stopped posting in 2010. That's the way it is, unsupported ideas like design ideas, that haven't been solved for many ideas, those ideas remain in the offing when the conditions aren't solved, they take the perseverance to wait until the circumstances collide. Luck is a coincidence (as philosopher Paul calls luck) of the right things colliding in the that point in time, that sparks wide acceptance of an idea that's arrived. Leadership of any idea can't continue on for free forever.

Burns III's observation that a lack of knowledge about licensing and copyright and fair use are a major obstacle for the status quo to change. Those hurdles have diminished with the advent of free software licenses like MIT license, Free Software Foundation Gnu Public Licence (GPL) and Lesser GPL, Apache licence, and so on in versions that appeal to many developers for the broad areas they cover.

Unlike audio/video, when a software house ceases to do business, the finished product can be decomposed from the program form back to the code. Songs and movies can be copied on for ever, no matter the media. Sure, there's an internet archive that you might find the files, but you're unlikely to find the source and documentation that was only stored on old computers. It took me a week to find and copy my old thesis copy I found on a 3 1/4 inch floppy disk set onto a USB stick using a 25 year old laptop computer. I'm surprised it still works. What is the motivation from former employees to search through files on ancient computers to find stuff for use by someone else? There would be motivation if there was salvage money in it. There must be a pay for play motivation to get people to find what a world wide salvage effort. There is also an obstacle in the other side of legal issues: how does one improve the downside risk for old code donated that removes legal liability from anyone that doesn't want the code? Google gave away the protobuf-c project with indemnification. How do you motivate goodwill donations with fear of legal repercussions. If you improve code and it breaks something, does that legal jeopardy involve just you or the original designer? This is an unresolved legal point. Another difficulty lies in the government's priorities. Legal and legislation would make some of these easier. There is less valuation of productivity when the costs are in the design of units produced. Developers could be compensated for the code, government can waive liability on former developers. Why not allow tax writeoffs for companies that donate code for public use as well as payment of a low Fair Reasonable And Non-Discriminatory (FRAND) license fee to the companies. The government has many ways to change the levers of the economy and evolve the status quo.

Budget and Schedule.

1. The best reuse scenario allows new designs to use the reused code by not delaying budgets nor schedules. This is the gold standard, extra value for zero extra cost.

2. Interdependence is a good concept to exploit. If tests made by one group solve testing needs of another group of developers, even when the first group of developers aren't even aware of; interdependence is valuable as asynchronous and mutually beneficial.

3. Programs officer's mentality of blue sky *carte blanche* desire to start from scratch is a hard desire to overcome. Professional management want to do things that impact their career, to be the first to create a novel thing, or building from the ground up is a far more motivating project than just working to eek performance gains out of many small things in a maintenance phase. It is the pioneer spirit and mentality that skews away from risk adverse methods to riskier ways that reflect on higher recognition and fame for the managers that want to win. Program managers would change their tune if the compensation penalized the riskiest behaviours and rewarded the most efficient ways.

5. Software not seen as an asset. The value of software would change if the programs in total revolved around the software and not the reverse.

4. Lack of Centralized Catalog of Assets Burns III How to define? How to advertise? How to pay for software reclamation? How to categorize?

Search applied to the code is the significant deficiency underpinning most code, forcing code stranding.

There are functions people want, there is code written to solve the need.

How does one marry the two up?

No way to test and accept the code isn't as big a problem if the testing, verification, and validation as NASA defines it becomes the gap to be bridged, at the end. Any repository that provides the code in a way that non familiar users can find, test and be satisfied; will be the way to finalize the novel search method.

Another grim reality. Most code isn't ready for public viewing, not in an organized and clean state. How can we motivate people to go back into the code and make the improvements needed for the mainstream to exploit?

5. Legal/ Contractor Issues

Until the liability / and license issues are solved across many nations there may be no appreciable change in the status quo of software reuse.

N.B. I will point out that even the reuse of this chapter, but copying out of .pdf files graciously and magnanimously donated by the US DoD publications process, it took significant effort to reformat these two chapter sections back into text format, requiring even a look at the requirements for documentation to exist in a reliable form, towards the reuse of all information hard won through shared experience.

Chapter 5

The ReUse Software Movement

Overthinking
Is The Biggest Cause
Of Unhappiness.
—Siddhartha Gautama, The Buddha

This chapter explores demonstrations of software reuse concepts, models, processes and repositories, in brief, to give you a quick survey of all software reuse ideas. Like every chapter, this is a work in progress and the

Looking back to look forward

In the late 1980's and 1990's there was a drive across many US Department of Defence organizations like US Air Force, the Special Project Office, Office of Navy Research and others to rework and remake software developed across many different applications but the same client into a transformative reusable libraries entreprise wide. There was the Software Technology for Adaptable, Reliable Systems (STARS) project, the Central Archive for Reusable Defense Software (CARDS) project and the GRASP (Graphical Representations of Algorithms, Structures, and Processes) project at Auburn University. Truly, the USA is such a large enterprise that I doubt all the actors knew during their work of the depth of parallel work in the meantime.

This ReUse movement went onto the demonstration and then... nothing.

These ideas, all reasonable and obvious in most cases, never did see any traction and I know why.

The proposers of CARDS repository had a vision of what the library storage, description, performance metrics, and knowledge base. And then the STARS, CARDS, and GRASP wrote up their final reports and moved onto other work.

Project teams, working on the actual systems went back tot he bench and continued to use whatever their company, team, or personally they had available to them. They had deadlines and demonstrations to achieve, and they were just as concerned about reaching these very real goals that mean a paycheck than the esoteric dream of reusability. Private companies that compete together for work weren't going to share for free, nor allow others into proprietary code - which might also mean system security to anything they shared - that everyone put down the dream that seemed impossible. They went back to unit and system testing to their own standards, and they tested as adequately as they could for the new products they competed to make. After all, they had their demonstrations to make. All of this effort, while dutifully recorded for posterity, died on the vine. Like many good idea that are thoughtful, optimistic, and altruistic there wasn't enough traction to make things move on their own. Great ideas, poor effort, and accepting early defeat.

On the other end of the spectrum, there were plucky, determined upstarts creating new source directories, and new software repositories, and the amateurs, neophytes and junior technocrats drive a mighty way towards complete working systems. They sleep, eat, socialize little for a turbulent campaign to make it work. And they it working to some degree. They announce it with great fanfare and celebrate the real accomplishment. And after this thunder clap, the darkness and dead silence. These accomplished few gain the accolades, write a document or conference paper, and either go onto bigger and better things. And, errors and all, all that work from every competitor that never got the contract, never went on to augment that work.

And then there's me, I searched and found a delightful software library collection repository. It seemed to have all, or most, the important things for a better way to communicate the software function and form. I knew it had a quirky (less common) name. It stood as a definitive and wonderful oddities. But, one problem. I couldn't remember the name. I can't activate the right memory areas to find within my own knowledge the name of the software archive. I can't remember that odd name. I tried many recollection techniques, I tried

Menagerie? No.

Curiosity? No.

Aha! Rosetta Code!

Here's the Introduction to Rosetta Code:

> *Rosetta Code is a programming chrestomathy site. The idea is to present solutions to the same task in as many different languages as possible, to demonstrate how languages are similar and different, and to aid a person*

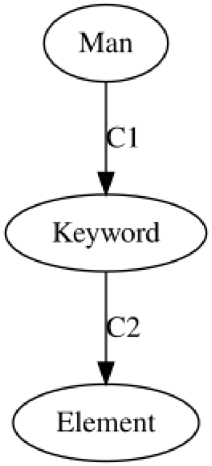
> *with a grounding in one approach to a problem in learning another. Rosetta Code currently has 1,181 tasks, 347 draft tasks, and is aware of 865 languages, though we do not (and cannot) have solutions to every task in every language.*[1]

It took me 38 (April 29 -March 23, 2022) days to figure it out. I ran across a hyperlink that ran backwards to the website. I am so mad at myself for not making more of the more to be made. Why didn't I bookmark it? How many people run across the word "chrestomathy" more than once in a lifetime? And that was the goal of the implementer, to make a memorable name, to stand out. I even recognized and made particular point of remembering this side-fact - associated with but not a substitution for -the library website name. And this forgetfulness of the obvious is something important and timely, because when writing a book particularly about software reuse, to have a unique name and for a software collection that was free and open source, and forget it, does a lot to underline that there are many problems making it hard for people even when people are not trying hard to make it. It's a huge flag to mark that the problems are real, and they are not easy to solve in one way alone, as can be seen by all the parallel stove-pipe failures that won't sink in. There are serious mishandlings of software even on the marketfront of the internet can make information hard to traverse.

But from this, I've made the singular discovery, by which I mean that in order to appeal to the needs of people that design things in depth, many pages of use cases in Universal Markup Language (UML), many facets of the same objects across differing design perspectives (hierarchy, data flow diagrams, communications, etc.) , and so on, that we must make everything at the single level. The distance from man to knowledge must be no more than 2 graph cardinality.

All functions must be one-depth from the reviewer, and all knowledge actions must allow the user to arrive at that knowledge from any point within 2 path steps or less. For those not as familiar with the ideas here I will make a simple category theory example. Most people call it Topos theory if they understand it better but I normally go no further than abstract categories because they are mathematical objects without a lot of restrictions on behaviour so they are normally a good starting place. Abstract categories are philosophy. Concrete categories are mathematics. Then after you understand better, you can delve in any direction you wish. From an abstract level there are only two things, two objects. There is data like a pear as a point amongst a set of fruits. And then there is a transition, an action known abstractly as a morphism. Acting on a pear

[1] *from website:Rosetta Code*



Graph cardinality 2- At no more than 2 nodes away from man for optimal human search

moves it into another data domain. But in concrete category the morphism is called a 2-object because it sits above the data on another "level". So cutting a pear makes pear slices - one kind of morphism - and smashing a pear with a hammer makes another data point to land at from the pear object to another kind of data range you land on. But the smash and slice functions are both defined on another level, they are data above operating on the data 1-object pairs. And so layers upon layers can be made, and are defined and held within toposes as pullbacks, pushbacks, and dualities, and on and on. There is a science of mathematics called category theory and while they understand things they are as equally lost as zoology because they cant encompass their knowledge into a reductionism like gravity as a field (which is a skew-symmetric ring in category theory) to arrive at the standard model of category theory like we have a standard model of subatomic particles that all obey QED - our friend Feynmann's quantum electrodynamics. It is my prescriptive criticism that at the age and awareness of the two fields, zoology and category theory / topology / topos theory, will both suffer in dark ages until they find their reductionism.

But, to make things accessible, the reusable library must be one level deep, it must as Grady Booch might call it the concept level. Functions and data must exist in ways people can scan through quickly, read a synopsis, and use. That's it. In order to create an acceptable interface we must keep it at the level of abstract mathematics. Because, as you have probably intuited from above, one can make an infinite number of concrete n-objects for all the morphisms that can be layered on and on. Inside the code, all objects- all concepts - will be

1-objects.

Now, that doesn't mean the library doesn't possess higher levels of knowledge. There will be layers of complexity, but that won't be inside the code, that's been the problem with all these efforts they've tried to mimic the complexity outside of human perception in one way. There will be dualities, there will be synonyms, homonyms, categories of kinds of functions. There will be ontologies, taxonomies, subsetting, and on and on. But the complexity will exist outside the code. Code will be small pieces people can cut and paste, or run with libraries configured.

Scalar Word Salad: Interpreting a Survey of Reuse Thinking

I used a survey of the literature from some of each of the software reuse reports, papers, and workshops proceedings on the matters surrounding reuse of software. You will find many references throughout, but in order to not apply my own biases to the subject (since I brought my own ideas about what works into my understanding of what reuse should be) I combined the text from all reports into one simple text-only file, then eliminated small words, proper names, and reduced words using background subtraction to compose what all these smart thinkers decided were the dominant words representing importance in software reuse. Now, there's a little bit of interpretation in this so I asked my self how to define the background. What I decided to use was that any word representing ideas that would exist whether or not any software was reused, so things like memory, hardware, software, firmware, store, biggness, time, locations, cities, nations, contractors, owners, and adjectives like always and so on were not directly relevant to the ideas of reuse. With this, I had 2MB of words. Cutting out words that would exist if there was or was not software reuse make it a scalar analysis to get to the heart of the matter. That's what I call background subtraction: it cuts out all obvious ideas that would be there with or without the core ideas. What must be left is central to the idea. I stopped when I had distilled to text down to 55,708 words.

Here's my rationale:

Bluntifying

Basically, any written work holds a bunch of important ideas, linked to important words by grammatical & syntaxical yet unimportant words, describing relative associations, done by people for organizations (we write to speak to people, not buildings, but we tell others which buildings we are in). My analysis is an analog of sensor fusion techniques: to background subtract all the words that would exist in the ideas before and after any software reuse, that would leave many concepts in their relative order for consideration as significant in the prosecution of the topic, in this case software reuse. So this analysis ranks by histogram the dominant ideas attributable by the largest population of words. Are the ideas obvious or relevant? That's for the reader to surmise.

The salary cap for all the people involved in these documents, from universities to NASA, was in the tens of millions, if not more. It's no wonder that many great minds come up with similar reasonable takes on the main ideas. They can't be far off about the general topics that should be included in better software reuse. But it's also a big trap of large organizations that sink into group think and tribalism like Not In My Backyard (NIMBY). The temptation is to reject ideas that aren't invented or supported in the culture(s) that writes these assessments. So my analyses looks into the scalar word histogram to look for two glaring things: the most common words and the missing words that they all seem to have ignored, overlooked, or discounted.

I call this kind of analysis bluntifying:

bluntifying the word histogram scalar product of documents's words reduced to the residual core ideas that represent a topic in the minds of the writers.

I am aware people have done this kind of thing before, but not perhaps quite like this, there are some heuristics applied to the interpretation to decide what is a topic that would be background like the idea of software whether there was reuse of software or not, and there was noise inserted by the optical character recognition that didn't create all the words perfectly, so there wasn't a clear list of data. It had to be manually considered. I erred to leave some words in that may be valuable even though they weren't on the face of it important. There were many hours spent vetting the data, removing the French "Bags" words (beauty, age, goodness, size) since these are descriptors that would apply both before and after, removing variant time concepts like now and then, removing conditionals like 'if', 'then', and so on. I imagined it might take a night and a day, and had to continue plugging away four days later, mainly on stubbornness.

Does this analysis hold value? On the one hand, this is a gross oversimplification, perhaps a figure has the word software 11 times, might that sway the outcome for the connecting words surrounding software(software, hardware, and firmware are all removed given they would exist with or without software reuse)? Of course a book of 270 pages may have an entire chapter devoted to the same concepts that link to a section of a journal or one page, the works will not be weighed the same. On the other hand, the fact that words appear more than others is significant because they deliberately chose not to include other words at the same time, or, in other words, any words signifying other concepts were not added were therefore deliberately left out. An omission or commission is for the final determination of the reader to conclude, and that's why I believe the work is valuable to you. You may see in the predominant words a meaning that assists you, that may not occur to others. These papers and books represent the thoughts of many many educated, experienced, and positioned people. It's no accident those words come up to represent the ideas and concepts over and over, it does tend to support their relevance in relation to the central topic. There will be many words that are just used to connect what these people think is important, take away all the "the", "and", "a" and other background noise of grammar, syntax, and style you reduce it to words representing concepts. And, can one argue that we AREN'T influenced by the words presented to us in a work as much as the hidden concepts the authors were trying to convey? I argue we do if we don't look for the hidden meaning. Bluntifying makes the interpretation strict and plain, more so than reading with our internal time-varying biases.

The single biggest criticism of this method, even as I finish the list off for the histogram, is how does anyone certify that when one person says construction in one work, how can we be certain that the same meaning is meant by another organization with completely different knowledge and culture? To this I reply: and how does anyone certify this in any other analysis, either? Mankind uses an imperfect language full of words with multiple meanings, to the level of roughness and uncertainty, that how does anyone extract any value out of any written work? A join in a mathematical sense is a product of ideas in various toposes, and completely different from the normal convention. You need to know a great deal of math to verify words are potential concepts even if the words may be innocuous. Of course, if you are reading this, I can be assured I won't have to worry about my losing out on the sale...

Some of these works were photocopied and the ocular character recognition program won't get all the words, there will be inevitable uncertainty.

So the naive inference is that the most words will correspond to the topics that these export BELIEVE are at the heart of the problem.

In my analysis, I spent hours reviewing, fixing the noise from the optical character recognition errors, and the

If you would attempt to do this kind of analysis on your own, you need not unless you have a person with a extremely extensive language knowledge, one that understands the idiomatic use of some words in one field that differ from the average vocabulary, no matter what the language. You need to identify words that might be synonyms even if they aren't practiced in many areas of knowledge. This is still epistemology as a problem of the widest possible association for allied concepts.

Of course, if all the talking heads are wrong, then the work is by itself moot. Perhaps, but the evolution to discovery of the essence or the pivotal ideas must start from somewhere, and where everyone else started from, is a natural start. With none of the associated associations alive because we scalar-producted the sentences to one dimensional ideas - is a great synopsis of what ideas are included and what aren't. It may show an important idea that's important by it's unexplained absence. If the previous efforts failed and these were the idea starting points, then it makes sense to examine what the causes might have been.

Tomorrow's discoveries come out of yesterday's assumptions.

At the end of using both heuristics and computer sorting I was left with a list of 1228 word histogram bins counting ∼55,000 words with a word count high of 1283 use for 'model' and a minimum of 5 uses for the lower threshold words like abilities, administration, and administrator. That was good to see. In this histogram of bluntified words and by inference concepts leads us to these rankings: Figure 5.2 shows the 1228 words in a histogram with 5 occurrences or more. The histogram is very large, but it obeys the Gaussian distribution, which was a surprise. Not an entire surprise, I always assume a Gaussian, but it was validating effect to see one.

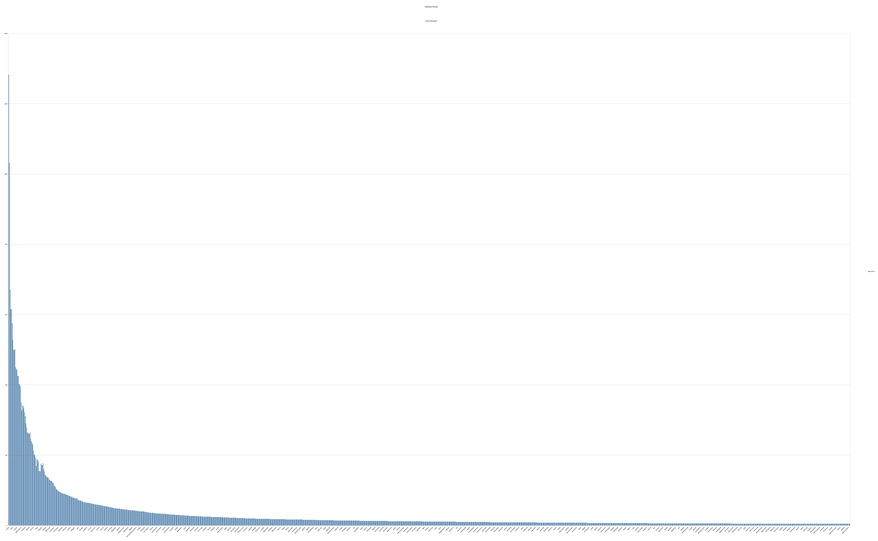
The top 10 words from all these texts about the software reuse topics are outlined in Table 5.1.

Word Histogram for all words down to 90 counts in Figure 5.3. The word counts down to 5 count, at which point words were considered noise, is in Figure 5.4. It turned out to be more work that a brute force chapter by verse comparison, but in a novel way it extends the hidden meaning into a common format. I will learn to automate it when I do it again, but grinding away at interpreting words made me conversant with the possibilities. Here are the three major words defined:

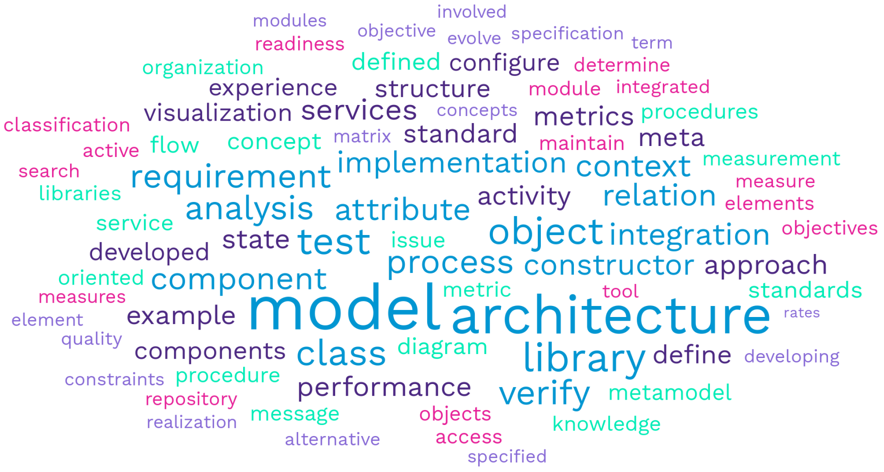
architecture (countable and uncountable, plural architectures) The structure and design of a system or product.

model (plural models) (software architecture) In software applications using the model-view-controller design pattern, the part or parts of the application that manage the data.

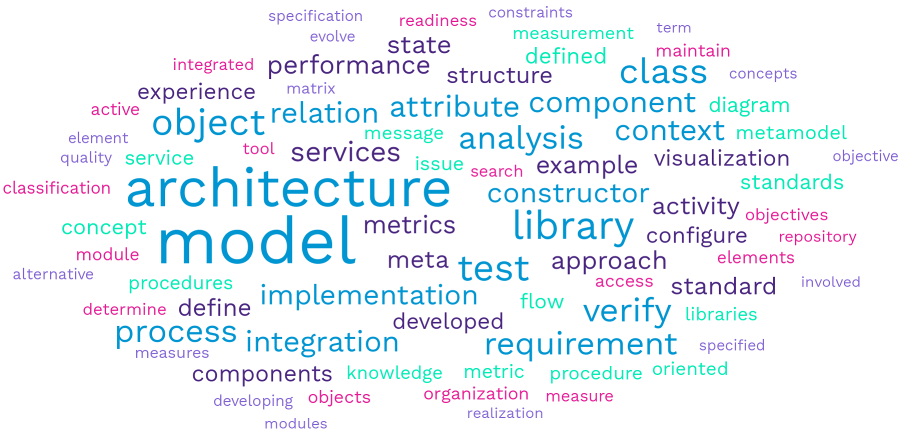
object (category theory) An instance of one of the two kinds of entities that



Software Reuse: Bluntified shows the histogram of the most salient 55,000 words from my survey of software reuse literature. The largest word count is for the word 'model'. While you may argue the value, it's uncanny how many educated, experienced, and well-funded software applying organizations think the same things are important.



Bluntified Histogram of words, top 100 from 1283 to 90 count words



Total Word Histogram (down to 5 count)

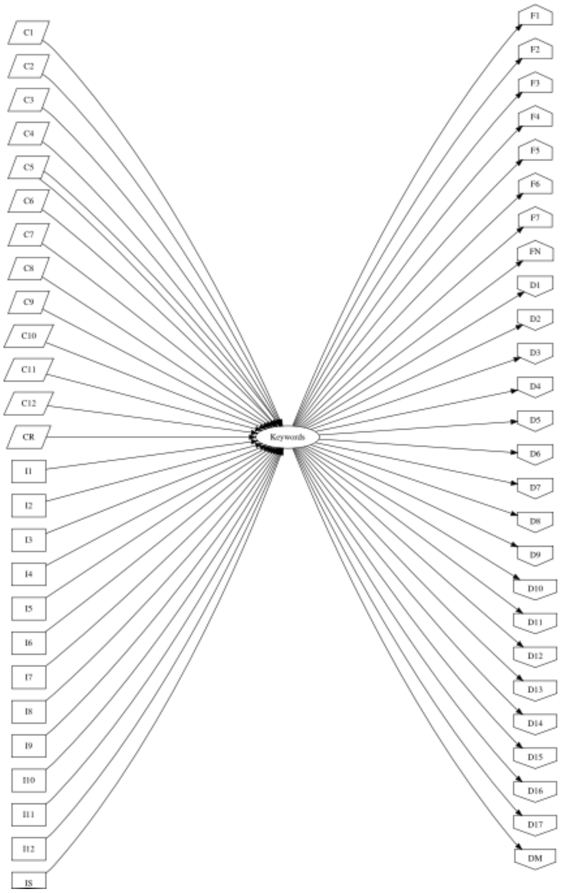
Word	Count
model	1283
architecture	1032
library	671
test	615
object	616
class	576
verify	528
analysis	501
requirement	498
process	502

Table 5.1: *Top ten words from bluntified analysis*

> form a category, the other kind being the arrows (also called morphisms). Similarly, there is a category whose objects are groups and whose arrows are the homomorphisms from one group to another.

So, combining the major important factors together and it's confirmed what people think is important, and what they haven't appreciated as also important.

From my analysis, I arrive at two conclusions: model is the primary idea encapsulated into the idea of reusable software. People need a model to cling everything to. The second is the general lack of recognition that keyword, the idea connecting idea that we use to bring many different people from various background into understanding ideas we can share, is completely ignored in the workings explained. A model is the way we describe a form like a skeleton



The Keyword Importance: Keywords link both Concepts 1 through R and Ideas 1 through S to functions 1 through N and Data 1 through M.

to others. But once they understand the general gist of the structure, they begin to look for the important bits to them. In this case, they will need an ironclad way to describe how TO FIND topics INSIDE THE MODEL. That's the biggest problem. Developers fixate on one and ignore the other, without exception. And we wonder why people can't find what they are looking for in a reasonable time. I discover that the association middle ground, as shown in Figure 5.5, is the missing vital mechanism.

My analysis draws a distinction of the consensus important factors, and the missed opportunity in keywords (which I expound in section6).

Not to divert fully from considering what people said, the next section outlines what I found to be were the highlights of the literature's concepts for reuse laid out in the next section.

Proposed Methods

Apart from my above-outlined math argument for what makes the vital importance

Basili: Quick Fix Modeland More!

Victor R. Basili proposed the following software reuse models[3]:

1. The quick fix model;
2. The iterative enhancement model; and
3. The full reuse model.

Basili's Quick Fix Model: [3]:

> *The quick fix model involves taking the existing system, usually just the code, and making the necessary changes to the source code and the accompanying documentation, e.g. requirements, design, and recompiling the system as a new version. This may be as straightforward as a change to some internal com- ponent, e.g. an error correction involving a single component or a structural change or even some functional enhancement. Here reuse is implicit.*
>
> Basili's Iterative Enhancement Model: [3]:
>
> *Iterative Enhancement [5] is an evolution, model which was proposed for software development in environments where the complete set of requirements for a system were not fully understood or the author did not know how to build the full system. Although it was proposed as a development model, it is well suited to maintenance. The process model involves: 1. Starting with the existing system requirements, design, code, test and analysis documents 2. Redeveloping starting with the appropriate document based upon analysis of the existing system, the changes through the full set of documents 3. At each step of the evolutionary process, continuing to redesign, based upon analysis.*

Basili's Full Reuse Process Model: [3]:

> *While iterative enhancement starts with evaluating the existing system for redesign and modification, a full reuse process model starts with the requirements analysis and design of the new system, with the concept of reusing whatever requirements, design and code are available from the old system. The reuse process model involves: 1. Starting the requirements for the new system, reusing as much of the old system as feasible 2. Building a new system using components from the old system or other systems available in the repository developing new components where appropriate.*

Basili's[3, 4, 2] models refine and restrict the ideas of reuse to the concepts that you understand

Tracz: Three C's 3C(Concept/Context/Content) of Reuse

Will Tracz wrote a software journal and then book[15] back in the 1990's during one of the revivals of software reuse, and then wrote a book on it. He was summarized at a workshop on reuse with his very inspired 3 C model(Concept/Context/Content)[AD-A226-985 REUSE IN PRACTICE WORKSHOP SUMMARY]:

> *The conceptual model for reusable software components was an outgrowth of the Concept/Context model initially proposed by Tracz in his dissertation work at Stanford. The model, referred to as the 3C model (Concept/Context/Content) is based on defining three facets of a soft- ware component:*
>
> *(1) The "concept" behind a reusable software component is an abstract canonical description of "what" a component does. Concepts are identified through require- ment or domain analysis as providing desired functionality for some aspect of a system. A concept is realized by an interface specification and an (optionally formal) description of the semantics (as a minimum, the pre- and post-conditions) associated with each operation. An Ada package specification with its behavioral semantics described in Anna is an example of a reusable software concept.*
>
> *(2) The "content" of a reusable software component is an implementation of a concept, or "how" a component does "what" it is supposed to do. The basic premise is that each reusable software component can have several implementations that obey the semantics of its concept. The collection of (28) stack packages found in Grady Booch's components is an example of a family of implementations for the same concept (a stack).*
>
> *(3) The "context" of a reusable software component is 1) the environment*

that the concept is defined in ("conceptual context"), and 2) the environment it is implemented under ("contentual context"). It is very important to distinguish between these two types of contexts because different language mechanisms (inheritance and genericity) apply differently to each. Furthermore, these two contexts clearly distinguish between type inheritance and code inheritance.

One can use type inheritance to describe the concept of a software component in terms of the operations and types found in another software component (what we are calling its concept). In other words, by using inheritance one can describe a new concept in the context of an existing concept. At the conceptual level then, the new concept "is a" specialization (subtype, or subclass) of the parent concept. Aggregation of concepts is accomplished through multiple inheritance. Parameterization or generality also applies to concepts, but its use is normally associated with passing data or furnishing contextual information such as the type of data or data structure being manipulated (operational context). In the 3C model, parameterization and inheritance play different roles at the conceptual level.

Code inheritance may or may not be used in an implementation. One need not observe conceptual relationships to access operations that may prove useful for the implementation of a software component. There are two separate contexts that apply to an implementation of a software component a visible context, one that the user can manipulate (operational context), and a hidden context, one the developer has chosen to use in the actual implementation (implementation con- text).

*Interestingly enough, both the operational context and implementation context present opportunities for variations. A software component's operational context is established by the user when, at instantiation or run-time, actual parameters are supplied for formal generic parameters. The implementation context is usually not visible to the end-user of a software component and established at build time. The component developer imports a specific software component or module whose operations are invoked by that particular implementation. But, given the environment defined by the 3C model, it is possible that several implementations could exist that satisfy --the semantic and syntactic properties of the module or component being imported or inherited by the developer. Furthermore there is no reason why certain aspects of the implementation context cannot be tied directly to the operational context. For example, if the user specifies that "fast, bounded" stack of integers is desired, then the stack package's implementation, might import a list * package that has been implemented as an array, rather than a linked list.*

> *One should note that while it is often the case that the concept and content of a component share the same context, the context of an implementation often subsumes that of the concept and * extends it with performance trade-offs, hardware platform, operating system, algorithmic, or language dependent contextual information. An example of a parametric conceptual context is the type of element to be stored in a generic stack package (an instantiation parameter). An example of a semantic conceptual context is describing a stack in terms of a deque where certain operations are renamed and others are hidden. An example of an implementation's operational context is a conditional compilation variable that selects between UNIX and DOS operating system calls. An example of a component's implementation context is the importation of a list package (which may have several implementations).*

Software as a Service

Software as a Service (SAAS) is the updated software model. As a service you modify an existing code base over time with the subscription fees and extended service agreements. As a service one must appreciate all the contributing factors that affect the value, the saleability, and the long term survivability as a going concern. Mankind is littered with people that failed to appreciate that wider requirements, that failed the software companies were doomed with failed software projects.

I don't argue that it will be entirely that it will be entirely responsible and committed to a single path for all projects when you write some code.

I argue that if you only look down at a computer screen, then can you really convince yourself that will be enough to make it last?

Did you think about all these side issues when you first started out? This book isn't written for you if you are planning on writing 10K lines of code (LOC). If you are going to pump out 50 LOC then this is far too involved.

It's ok. Don't take these words to heart. It's not meant as anything but a heart-to-heart tough talk in an elevator from one floor to the next. On the way to a coffee shop to caffeine up for your next blitz session.

You'll live.

But think about it. Wouldn't you want to make your life easier by doing so well at this software ecosystem gig that you never have to worry about other work to support you and your family?

When you were a university student, you can demonstrate your technical expertise, project experience, and even some human soft-skills/ people skills.

It's a way to cut your teeth as the expression goes.

You make some source code, fix a bug, make a key design change, port some code for a custom chipset, or device use, and in exchange you can claim your participation in a real live software job.

I've often thought that if a concurrent version server (CVS is presented later in 12) system had an automatic letter of reference generator included with all committed code that it can describe your statistics from your additions to the database, and it creates a custom, fact based "report" on your:

- numbers of lines of code improved;
- numbers of bugs fixed positive comments; and
- awards and milestones completed with or without supervision.

It would go a long way to describing a potential software developer hire to a committee with the merits of an individual in an objective manner far better than a polite but cryptic letter of reference or a mundane curriculum vitae and resume.

I predict that, within 20 years, that this will be the objective standard for anyone wanted based on software skills.

Here's the hiring truth:

Skills exist outside of education, knowledge, and experience. What an employer really wants from someone isn't a short list of accomplishments and a long list of degrees, course titles, and awards, if any.

They all generally make the short cut into accepting any university level accomplishment that represents the fact that you can learn, even at an elevated level. It may or may not be relevant for most jobs how much your knowledge background is. I know that unless they want some really obscure set of knowledge for the job, like your Ph.D. thesis, then any knowledge will do.

So, back to my original question as a valid examination of your ambition, your career, and a software project.

Wouldn't you want people to accept you for a known accomplishment, one that you've poured your extra time into, and use that success as the basis of your commercial worth?

Linus Torvalds is a multi-millionaire based on Linux. He's world famous even if it's just the closet nerds and geeks.

Even if you want no reknown, reputation, or glory from your code, there are no better ways to make a professional impression about you (and your accomplishments) than how thorough, complete and professional you operate a software code base. It's a best way to make that impressive aura to you and your future.

Back when I started coding, it was important to find investors, make a sales pitch, and capture capital, and then figure out how to get to market and somehow get finished. Late night pizza sessions. Missed birthdays, missed class reunions.

Today, people complete software to relax, to make a social statement, to record their cats making some weird art, to act like old hackers making a software defined radio in their garage out of vacuum tubes and old soviet missile components. Software is valuable, but code is now cliche. It's quirky, It's off the beaten path. It's in everything we use and it's no longer special and unique in any way. So one must adapt to the new reality. Instead of using one cylinder of an engine to drive forward, you can find and harness many kinds of adherent pistons to drive.

For Linus Torvalds, his adherents range from religious acolytes like summer computer science grads, to student undergrad groupies that fall out of interest when economics has better starting wages. He has leveraged the software he guides and adherent's workings to push the software juggernaut that Linux has become. And will continue to exist long after Linus is no more. Into future projects, into retrogressive projects for historical retrospectives. Even if the Linux project falls into disrepair, even if the original code is lost, even if the adherents all drift away, there will be amongst the discarded junk some remnants, somewhere. Even if it's just a dusted file folder lost in a filing cabinet. Some old version manual will make it's way into someone's hands. This person will read and heed the wisdom within. This person will go onto piece together large portions of the work, understand the rationale, the file directory layout, and get excited about all the possible wonderful applications that can be again if only get it all working. And from that one reference, like the lone book of holy scripture in A Canticle for Liebowitz, that's all it takes for a person to dream and create anew. From all the value that was stored in Linux, which was estimated many years ago to be over $1 billion USD, that it's all that stored value that has what people want, it will draw them to it.

That's really what, from a cynical perspective, is what great valuable things possess, it's the way that they allow people to get more without giving a lot, there's so much more in adopting something 80% good enough than rejecting it and starting afresh. It's what draws people to be lazy. Tools are useful, like a hammer. You find your hammer when you need it, you hammer what needs to be done, then put it away. You don't saw a limb off your tree and whittle it down if you can't find your hammer. You might look some more, or you might go to the hardware store and buy another one, determined not to lose this one[2]. Once you are finished with a hammer, you put it away. Linux works the same way.

According to Milan C. Babak of IBM, the longest software project in history is the IBM Z mainframe development. It has been operating since 1964, that's a whopping 58 years. It's claim to fame is consistent working for 24/7/365 operation for large businesses collecting transactions, Based on the design principles of backward compatibility and effective user interface, it's been the longest and most dependable point of sale (POS) for big names you know like Walmart, Banks, and Airlines.

so, if you look at the two largest software projects in history, the IBM Z mainframe and the Linux kernel, it's clearly that both project exceed $1 billion USD developed value, have several hundreds of thousands if not millions of lines of code, and long histories with many adherents. These are great upper bounds on many business in software ambitions. Many companies struggle their whole lives, their founder's lives, without achieving the $1 billion USD unicorn in value status.

Of course, by virtue of their relative positions in their markets that they inhabit the pole positions and with good reasons. Point of sales software handling hundreds of thousands of transactions per day, and the operating system within a majority of servers worldwide, running many other operating systems within virtualization containers on top of the core kernel with billions of servers worldwide.

At this point in their careers, they enjoy both application pull (a market demand begging for more) and technology push (novel technologies making incremental improvements spurring value) in their respective fields. If this is the model that ensures success, to imagine the perfect need - the so called killer app - and the deliver a good starting point software code base, on top of reasonable hardware, and continue to evolve and improve the system writ large over a continual basis, to examine, reassess, and plan to optimize where the needs are, to include new features and applications, and commit to perpetual improvement despite setbacks, failed attempts, and delays.

Daunting as it may be, this is a guaranteed billion dollar value software entreprise. The goal of anyone striking it out on their own with information technology.

There it is. This is the proscriptive solution to your needs, a gold mine in wait of a miner. Hard work, and this proven strategy will attain, in this time-tested

[2] I have duplicate, redundant hammers all over my house like I have three copies of Green Day's Kerplunk in my CD collection

way, proven for over 30 years despite massive hardware and software changes, to continue to pull in the money and reinforce efforts year after year.

If you ever wanted the keys to a software kingdom, this is it. This is a legitimate way for a legacy. All you need to do is to figure out what untapped potential you can provide, what will drive you to succeed, and what unmarked territory you can exploit. With this in place there is nothing stopping you from spurring your way to victory.

Back to the ideals of software development.

In 2008, it was estimated that there were a doubling of existing software projects every year. So, as well as the inspiration to discover the next great application, that in order to go fast enough to meet your ambitious timelines, one must possess the skills to find, understand, and harvest/cannibalize/repair/diagnose/adapt and leverage (by leverage I mean exploit the work already made) these projects.

Here's what these mean:

Repair -

make necessary changes inside the codes intent and mechanical functions as is. This means to make the most of it you need greater understanding because for many instances you will need to watch more than one piece in action, that means unless you are lucky at what you look at first, this will take significant work.

Harvest -

Minimal interaction with the entire code base, you find something interesting and you extract it. You are taking half of something, not the You can make a linear relationship between what you read and what you harvest. You don't need the build architecture, the global variables, as little as you require.

Cannibalize -

Pull working parts intact working the way they are and you find them. Your learning about the code stops at the border of what you make.

Diagnose - Look into the workings of the *in-situ* code and understand how it's working and how it's not working. Deep dive (I hate than euphemism because no book acts like water) and you get into the minds of the developers.

Adapt -

Adapt is the happy medium between ripping stuff out and fixing things on the inside. In this case, more of a grey area than cannibalize or repair, in that you take some parts out the way they work, and you include them by adding them into your own code. In this way, you are changing the function and data of both to make them fit together.

So, like the complexity versus size graph, the amount of work is commensurate with the amount of stuff you plan to encompass. If you take out a piece whole and don't look at the innards until it crashes then it's a risk of not completely understanding what you've installed.

There's an entire industry for people with the problem of not having reliable parts and supplies. Think aircraft carriers made in the 1960's. The UK stores a whole aircraft carrier in a river inlet just so they can cannibalize parts for the remaining one. It's called diminishing manufacturing. If you make this your business then you will be in the diminishing manufacturing game as well. Any obsolete and end of product run, like many dead software projects, which will be the majority of all software, that has outlived it's usefulness to the owners it becomes part of the scrap heap for you. There are always gems amongst the spoil. The facts of diminished manufacturing is that you have to learn when to stop more work into tested and flawed code and when you are wasting more time and effort that would be better spent looking in some of the alternatives. There is a learning point where and when to seek value elsewhere, be it due to a lack understanding, the code's failing are not in your wheelhouse, or the long term benefits don't outweigh the sunk and extra costs to make it valuable. But these are the obvious but not the only ways to use someone's code.

There are many ways to use someone else's code other than the straight assimilation of it into your needs. You could do the following:

Make the other software into a cantrip; using the code as is, and understandings it's inputs and outputs only, compile it to run as a special set-apart program. Like the Unix philosophy supports, pipe into it or from it data that you need for something else without directly accessing nor caring about the internals.

Make it a Rando

Compile and execute the code to just sit there and do whatever it does. The idea here is you don't even care to understand what it does, you just use it as a generic load that takes up space and resources within a working processor. Think neutrons in electric fields. It's there, but so what.

Make it a security target

if you goal is to use it as a target for denial of service attacks, then you are better off trying to attack it running first without the cheat codes of understanding it’s internals.

Make it a stand-in, or standby

Knowing some of the functionality and API, use it as a temporary part of the system (again without meddling internally) while you are examining computer activities at the level of messaging systems, middleware, and so on. If you are locked into a perpetual maintenance cycle, this can be a quick workaround.

Chapter 6

A Reuse Strategy beyond Fads

He Has The Most
Who Is Content
With The Least.
—Siddhartha Gautama, The Buddha

The first question you may ask yourself when you read this chapter is what for and why is an author bothering software manager types with some idle math presentation. Simple. In order for you to manage your software future prospects you need to understand at the highest level from an objective perspective what should be happening at any level of your software. If you can adopt and use a rigourous math argument, then you can measure, analyze, and judge whatever software dog's breakfast your coders come up with. You're welcome.

Erickson Composition

We need to mentally layout a grounding philosophy to describe the better way forward in mathematical terms. We define idea and concept as fixed things for the purpose of explication and explaination. Every philosophy gets grounded in math. That's how philosophy becomes (somewhat) useful.

I define a way to link keywords from things like functions and data within a library to a generic set of concepts and ideas that any software developer might use or understand. Keywords in a knowledge base (in this philosophy they are internally connected apart from the library in question - we do this with conceptual graph interface format (CGIF) developed by John Sowa inside RULA).

idea[1] (philosophy) An abstract archetype of a given thing, compared to which real-life examples are seen as imperfect approximations; pure essence, as opposed to actual examples.

Let all ideas related to software be known as nouns, places, patterns, objects, archetypes, examples, somethings, someones, somewheres, somefors, arrived-ats, destinations, sources, domains, ranges, co-products, 1-objects, and so on.

concept[2] A description of supported operations *on an idea*[3] type, including their syntax and semantics.

Let all concepts related to software be known as verbs, actions, someways, abstractions, somehows, arrivings, operators, products, transformations, functions, functors, pullbacks, 2-objects, and so on.

There is then a product of the mashings[4] of objects and concepts (a multiplication like a cross product not a co-product like an addition). I use a modified ncatlab.org definition for compositionhttps://ncatlab.org/nlab/show/composition:

> *Composition is the operation that takes morphisms $f : c \to i$ and $g : i \to k$ in a category and produces a morphism $g \text{ə} f : c \to k$, where* ə *is called the Erickson composite of f and g.*

So, I restrict the idea of abstract morphisms (a vague idea about actions) to the idea of compositions in making software keywords. Is this reasonable? Who knows? It's a start at a beginning. And, I introduce the Erickson Composition for Concept-Idea Products: In order for any idea I to be found elsewhere (for example, idea i_7 found at the same object location as the object k_3 then i_7 must be acted upon by the concept c_1 onto k_3), then the composition is shown in Figure 6.1.

The Erickson composition equation is then:

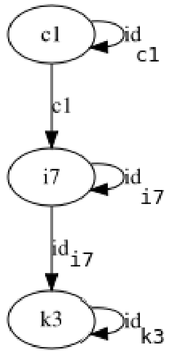
$$c_1 \text{ə}\, i_7 = k_3$$

where ə is the Erickson composition morphism from one concept to one idea.

I restrict the domain of f, to the identities of every idea in your head. Not mine. I restrict the destinations of composition to the range of keywords. Not

[3] my modification from original

[4] left undefined for the purpose of your use of whatever intermediate concept that suits your cognition in pursuit of proscriptive advice rather than strict mathematical proof



Erickson Composition: The Concept acts on the identity of Idea so that it can be found within the keyword object

my keywords, yours. If you understand none of this just nod to yourself that I have made defined linkages from ideas to concepts. ***QED***.

Remember, the ideas, concepts and keywords all rest within your mind (the software) and your brain (the hardware). The functions and data within a software library lie inside the library and it's documentation as the authors intended, or not. The goal of this chapter is to set up an understanding of the mapping between unknown work inside software and the things you hope to find (or not find) in that candidate for software reuse.

Let **keywords** be the product of an idea object's identity (i_{id}) composed with a known concept. For a generic composition, let the object of the product of one concept with the idea's identity be called a keyword.

$$k_{3id} = id_{i7} \circ C_a = id_{k3} = c_1\text{ə}\, i_7 = k_3$$

where k is the keyword, the identity of the object id_i is composed with concept C_a to arrive at k_i.

This is what this means, and most software is already organized like this: If you want a GPS data struct initialized in some code you don't understand, then you are looking for the generic keyword for GPSInitializeData (Words for concepts and ideas smashed together like WordsSmashedTogether) or perhaps just GPSDataConstruct or ConstructGPSData. That's the basic syntax for keywords of the Erickson Composition: you find the ideas composed together into an identity that is unique.

Now consider an unknown software library you are intending to leverage, and you wonder if there are or are not specific functions (functions not in your mind but in the mind of the original software developers) and specific data types (data types not in your mind but in the mind of the original software developers) that can or cannot fulfill your intended purposes.

The case for finding out that a library is without the ideas and concepts you need is a mirror of the argument I will now present. Just imagine every object and concept I present in the negation form ($\neg i$ or $\neg c$). In truth, you will find that most software libraries hold both the ideas and concepts you want and also not-ideas and not-concepts you want or don't want. For the purpose of clarity I only deal with the existence of things to find, not with the non-existence of things to not-find.

Let X be the range of functions or object methods that you want to use or leverage.

Let Y be the range of data types or things that you want to use or leverage.

The first realization you make (without presenting any equations) is that you will need time/resources invested to find out of there are all, some, or none of the functions and all, some, or none of the data types you want. Any leveraging apart from copying will lead to a gray area of what things are useful and what aren't as they are presented. This is where managerial judgment comes in.

These are the **five situations of code reuse**:

1. No functions and no data you want to leverage
2. Some functions and no data you want to leverage.
3. No functions and some data you want to leverage.
4. Some data and some functions you want to leverage.
5. All data and all functions you want to leverage.

The most unlikely and trivial situations are 1. and 5.. It is unlikely that you will investigate an ice cream control software library and find nothing related to ice cream control. And the situation of the library has all preferred functions already in code with the perfect data types in the preferred format is also ideal but unlikely. The realization for any of the situations is the effort needed to determine an outcome will never be zero. And the cut-off point for too much investment with not enough return is a problem. It's a paradox in fact: if you can't be certain that one extra second of investigation, one extra dollar invested won't determine the certain outcome of the desired function or data exists or doesn't exist, then how can one determine the value spent as loss or profit? This is like the **sunk cost fallacy**, but in reverse: it's the **reward certainty determination**. In other words, at what point can you get to that changes the value invested from a loss into a certain money maker? There's no simple way to calculate it, nor definitive time condition to end it. I leave the creation of these measures to economists or anyone else that uses numbers to feel better. My objective of pointing out the problem is the bound I choose for this work.

Situation there are fewer y's than x, and you need to search all x's in y to find the ones you want:

$$Let : x, y > 0, y < x, y \neq x$$

Let A be a set of topics the programmer is interested in. Let B be a set of smaller items from which to find those items in A (B is a subset of A, A is a superset of B).

The search space for any x, has a best case of 1 search (the first item is $x = x$), and the worst case is y (the last item y is the x searched for). And if the specific item you are looking for doesn't appear in the searched set, then with no memory, that is like you searched the whole space for nothing. That will be the same for all no memory searches which is the worst case scenario.

So, assuming that all identifiers are equal to the searcher's version understood, there will be a total bounded number of searches as $[x, xy- > x^2]$ assuming no memory on searches.

Lets make the problem the opposite scale:

$$Let : x, y > 0, y >> x, y \neq x$$

Let A be a set of topics the programmer is interested in.

Let B be a set of larger items from which to find those items in A (B is a superset of A, A is a subset of B).

The search space for any x, has a best case of 1 search (the first item is $x = x$), and the worst case is y (the last item y is the x searched for).

So, assuming that all identifiers are equal to the searcher's version understood, there will be a total bounded number of searches as $[x, xy]$ assuming no memory on searches.

This is a far bigger number of searches, because y is much larger. These are linear searches, like looking at a line of people and checking for one face (like Bellman's original theory).

The way to speed up hierarchies to search is to make complex search trees as they call them (this book is written for spunky neophytes as well as computer science experts) that divide the search sector into smaller and smaller groups so you don't have to look at all faces in the line from one idea or concept to the function or data you are searching for. And many people have done so by implementing that complexity ONTO THE LIBRARY of software.

This is what this book has lead you to. The pot of gold at the end of the rainbow. The rainbow leads to an easy search and easy implementation. It's not that this single layer isn't easy to implement, but that people were making things unnecessarily complicated by organizing, introducing complexity that may or may not mean anything directly to the operator looking for the one thing they need. That's why that complexity can't be in the code. By making internal decisions about what to include, what to show and what to hide, we are unnecessarily making it harder for people that won't use that particular set of concepts to arrive at the one they want. Over complicated makes obfuscated. In the end, the best starting position is to look at the index, not the table of contents. Or, the table of contents but not the index. And so on.

So, what this means, we must allow for all the search, index, and other concepts but make them particular to the user.

Let's look at the complexity problem from the standpoint of multiple identifiers that each are surjective onto the limited array of concepts within a library in finite storage and finite functions.

Suppose that r represents an index number in set R for classification like a mathematical society's "Mathematics Subject Classification" that is a made up number like the Dewey Decimal System.

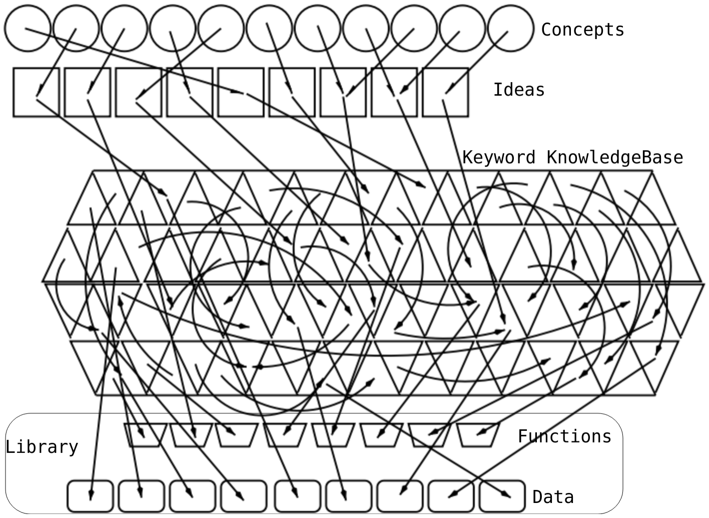
Suppose that there are concepts c in set C that represent main numbers associated in r above and would cover many sub topics in the "Mathematics Subject Classification" from zbMath.org. I picked this at random and not with any bias at all. So c in C are surjective onto r in R. That means there are more c's in C than r's in R and they all map onto zero, one, or more r's in R.

Suppose that there were keywords k in set K that represent keywords that are in c above and are therefore would cover many concepts in c of C and classification numbers r of R. So k in K are surjective onto c in C and r in R. That means there are more k's in K than c's in C and they all map onto zero, one, or more c's in C (which map onto r's in R).

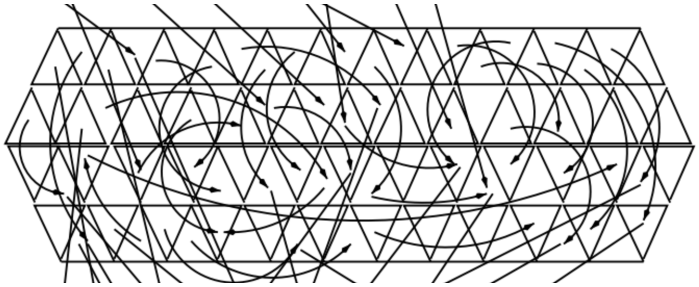
I will stop the math here for the managers, they will fall asleep. I will add it back in for a later version. For now, let's assume this is all true (it has to be done in CGIF which will require logical statements beyond a set theory perspective).

BORING MATH HERE

This produces an over-connected (as in many things pointing at one thing from an arbitrary number of sets) rather than a one to one or minimal connected set of concepts to ideas, ideas to functions, ideas to data, concepts to functions, and concepts to data. So, the idea is that with a search using many kinds of conceptual indexing, there are more chances that all it takes for one person to



Simple, over-connected keywords from many sets of concepts and ideas onto one function and or data inside the library under investigation. In the large scale



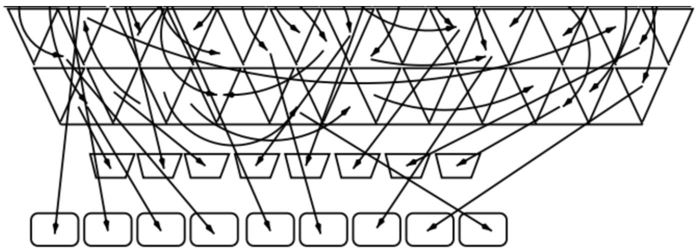
The impact of keywords, that multi-connected keywords that activate other terms.

find the right ideas by any selected concepts is through intermediary keywords, therefore allows more mappings onto the one concept that the coder wants. There are many concepts pointing at one data point within the set y of Y.

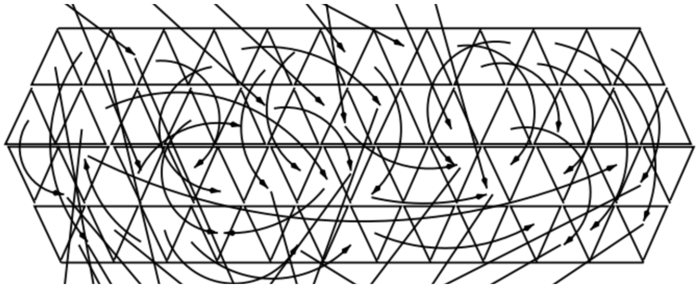
$$Y_? \cong (R_a \cup C_b \cup K_d) \leq X^2 \leq XY$$

So to add together a union of three sets is in general less than the parabolic of X^2 and polynomial XY. The importance of this can't be exaggerated as the set sizes grow larger. Consider this in comparison to a library that grows more complicated over every layer

Now imagine that instead of 3 factors, you have 20, 50, or 100 defined as mutually exclusive concepts online that are surjectively mapped to the functions



The Ultimate Goal: keywords allow multiple ways to connect to what the software coder is looking for without knowing all the correct, precise, ways to search for it. The user doesn't need exactly one correct way (in the understanding of the original coders ideas and concepts) to find what they need. The new coder will get a range of answers that might point at the wanted elements. Or, just as useful, demonstrate conclusively the library doesn't have what is needed. Quickly.



N over-connected ideas from many sets onto one brain concept.

and data at the bottom in y of Y, and a person connects from inside their brain's version of x in X onto any one of c in C, r in R, and k in K. Some of the concepts may be negative morphisms, as in a not-associated not-belonging to concept as a further way to allow a user to explain things their way. This is a totally new way of examining the problem, insofar as I am not using Bogart's Twentyfold path ideas nor Rota's Twelvefold way of combinatorics. I don't care about the search space if a human gives up after one or two tries. From my history of lost cause software projects, I attribute this, a readily and easily searchable way to find what they seek, as about 80% of the lost cause.

Above, we expand the problem to ideas that will influence the first search as successful, as many as possible. I call this way over-connected; instead of 1, 2, or 3 chances to get it right on the first try, you might have 100 complimentary/anti-complimentary ways. And after a couple of attempts, you either master the technique or give up. Or you read a tutorial and watch a video of someone doing it easily and that person is into the fold. I don't expect everyone to be a convert but if you can keep the adherent class growing as you lose a few, you will supersede the majority mass to reach a billion dollar software value.

This doesn't mean one or a few humans can set all this up. Setting all this up would be a Herculean task for a man to create. And that's how Sowa's existential graphs work their way in. ISO/IEC 247707:2007 is the way to embody layers of complexity without adding to the stratifications of the code itself.

If you want to take part in my continued experiment about the importance of precise keywords, this book is an experimental offshoot. At the end of the book, I created an all-concept index, rather than a separated author index, concept index, and symbol index located here Appendix B. Naturally, the makeindex LaTeX style file can create a consolidated index because it makes key tokens in any case. It is human beings that separate them for stratifications and order. If you are thinking of any idea presented in this book, you can run a mini experiment by looking in the index for concepts associated to what you remember. IF this index works better than the usual indexes, that should stand out as reinforcing the importance of specific concept-idea-keyword associations that last past reading something.

I cover Sowa's existential graphs in detail in the RULA library later on in Chapter 10.

Software As Essentials

Here are the four main intellectual components of any software/firmware and in some cases from certain views hardware (if one allows for electrical analogs of software concepts):

- Form: The layout of the things in the component, from different files to the configuration system can be quantified;
- Interfaces: Like all components, these components interconnect to other components like the operating system (*e.g.* Linux operating system function calls like fread() and so on)
- Data: the size, type, and volatility of the data that gets worked on by functions within the component that interfaces to other components
- Concepts: Every software component includes the concepts that the software developers had in mind when they wrote it from project requirements onto product specifications. Ideas like publish and subscribe for communications, read and write for data interfaces to the operating system, and so on.

If there is any way to inquire into software and reply in a way that will make sense consistently, then lay out the above form terms on a piece of paper, review the component you are looking at, and then fill in these 4 essentials and you will have described most of what you are looking at. When you have defined these essentials, the understanding that comes with that will make changing, modifying, and improving your software even better. your workload less onerous, and your plan to complete far clearer than any other critical thinking review.

If you take away one concept from this book, to divide software you are forced to review into four stark ontologs as above, you will have sped up your softwware reuse journey by many steps.

See Others in Your code

People will not like your work along one or more dimensions; how it's built, how it's organized, and perhaps even it's documentation. That's the first most obvious acceptance to the reality of making it easier to help other people: there needs to be a way built in that makes it easier for others to HELP THEMSELVES inside your code. That's the first proscriptive lesson from my many searches through over 5000 open source software projects. That's the first and most vital way to improve ease of use.

This book accepts the realities that people won't be happy with the way it is, but can be made happier if their transition of your code follows an easier path than the current one, using traditional ideas of what makes a software code base useful.

With that in mind, doesn't it make sense as you go along, as one can, to also make it easier for others as you make it easier for yourself, since everyone will be leaving software for others to improve upon?

What's your value proposition?

What will your value proposition to new, future programmers in this new century?

It's not syntax.

It's not grammar, nor rules, nor code coverage.

It's not cutesy names and TED Talks.

What you will win over observers and neutral parties like global corporations is with the knowledge contained in this book that:

- *Builds insight into the future of your software.*
- *Convince others to stop competing and begin collaborating.*

Your better plan that plots many paths forward to: Greater ease of use; faster code; more reliable because it's beta tested and bug fixed worldwide; more diverse clientele; a time-tested way to outlast that first burst of enthusiasm that accompanies big new ideas and new workers; Keep a common understanding; and and, ideally, making your code the most re-used in history.

Shifting people from neutral observers to followers - or more generically adherents - from a sheer numbers perspective, is indeed a positive outcome of a, from a macroscopic perspective (making your effort more significant), an intersectional perspective (your ideas cross more people's minds), and environmental perspective (maximizing resource usage and minimizing waste). If you tell people you are reducing project risk by dividing the work, decreasing risk of milestone slippage, and the ecological benefits it will appeal to the economical, the business minded, and ecologically friendly; you will stand greater chance of people supporting you through marketing, through donations, and yes even work.

There's an even bigger untapped market for science; science fair projects. If you can get a group to plan out and leave small "bite-sized" projects that are within the capability of a high school student, and then provide periodic supervision with meetings over the school year then this is an entire extra population to leverage. These projects can include difficult projects that win them scholarships, then you will make your impact far wider with national science fair recognition. I was the chief judge for a local science fair and supervised and coached teams of science fair students at national science fairs. I've watched our students walk away with $30,000 in scholarships. Even the younger ones treasured the experience. This leverage will bring you more impact, especially from the parent class, if you can cooperate to help high school students. And it might even attract corporate sponsors. Many large global corporations invest heavily in science fairs of one kind or another.

All these simple examples of looking at what you are doing from a few steps away from the code, away from the desk and chair, from a viewpoint I call the "holistic viewpoint" on the needs of software development. This is very hard for people starting right out of university. We sit in classrooms of raw science: calculus, complex calculus, finite element methods, logic, topology programming, metallurgy, and so on being marinated in objective reality, all of it absurd and beyond the grasp of most humans. But after tat, we strive to communicate and find common cause with nonscientific people that aren't aware nor concerned with any of that. How does one "bridge the gap" for an unprepared bridge hole?

If you watch the advertising of the global multinational technology companies that sell technology, their face to customers is entirely "soft skills" of normal human advertising. They've transformed and adapted over the years from accenting the latest tech, mainly from year after year of failed pro-tech advertising. The advantages are twofold: they appeal directly to normal people's emotions, to non-technical people, and the vague mentions of technology features are mainly deceptive and restrict or hide from public viewing the weaknesses and failings of that current design; and delays the realities of these aspects until after the sale. Welcome to the world of $billion advertising for $multi-billion global sales!

"***Sell the dream, refund slowly the disappointment.***"

Add to the fact that these same companies are known to tamper with their own products in order to goose new sales. This is the life you want, apparently.

Learn from these people, successful campaigns and companies, Take every aspect of your project seriously. Include the aspects that other people care about. This is how you win the widen the appeal to get co-investment. In some ways, these soft skills priorities outweigh the reuse of software projects as a pivotal improvement. It recognizes, in parallel, future needs as well as the immediate technical needs.

The bigger problem when renovating code is the nonunderstandable decisions with no documentation that gives a first time user a bad impression of the code and the coder. It's not enough to make the code "correct" (to the syntax, grammar, and format) and "complete" (that it contains all the memory and functions that you believe the code needs) as if those barely acceptable math notions are sufficient (mathematical sufficiency - it MAY be so) is when you leave no rationale then people's misperceptions will judge your code unnecessarily lacking (mathematical necessary - it MUST be so).

"But it works" is no excuse. But If you are not willing to heed, answer and implement all justified changes. improvements, and redesigns yourself (which can be very instructive to others that compare and contrast alternatives) then you expect the effort of others to improve what you have started. You need buy-in at some point. In fact, no matter how well you believe your code and coding style is, the time-tested demonstrations of commitment to other's software is how long you can keep people helping your code.

When people invest in your code, they are acting in their best interests as they believe helping themselves. That's the singular, time-tested, and proven performance metric.

There's no guarantee that others will continue on they may take on a forked trunk code, erase what new fangleds you've added, and keep it at a paused level where they both understand and accept what the code does and what it doesn't do. To each his own.

But when the code base starts with much fanfare with much fanfare, high hopes, and wishful thinking.

But check back in three years. Many code bases begin with lots of hope. You will see many dead projects.

Here is the software reuse *vicious circle*[5]:

If people don't know about your code, they won't use it.

If people can't use it from a simple high-level API, they won't include it.

If people don't include it, then they don't talk about it, mention it to other friends.

If they don't talk about it to others, those valuable personal referrals don't appear.

If people don't refer to it, then you will be forced to make all improvements.

If you make all the modifications yourself then your life be altered and your software will consume your life, your extra time, and your hope.

If you lose your hope, then your project dies.

Then you will give up.

If you give up, then your code may never be considered again, by anyone.

Programming languages come and go. Details of what you did and why will fade. Code is the basis of software, but code ISN'T your software, it's merely one facet of your software. Your software are the ideas and how you present them.

You are even in the same situation as for your code. You are not just what you look like, smell like, and act like.

I have written code since 1992, at least good code. I wrote BASIC back in 1988 very badly. I look back to my old code, at my master's thesis code which was written in back in 1999-2000, which was written in C for the Microsoft C, and I can't fathom why I made the choices I did, the style I used, the rational of file names and so on. As you write more code, and especially when you read better code, you will improve your style. Your programming knowledge is time-varying, why not accept it?

Myself, when I am trying to augment code inside someone else's code base, I adopt their style. I try not to destroy their style too much. I try to adopt a light touch. The goal is to make it less confusing if they find my edits and

[5] I wrote this BEFORE reading Captain Donald F. Burns III's thesis. This is from my experience with 5000 code bases

modifications they would be more likely to use it. Have you ever coded anything while thinking about someone else using it 100 years from your typing?

By adopting idiomatic styles, and different file layouts, it gives me a broader understanding But what makes me improve?

What is in the comments. Comments are the chain of logic outside the code, not the code itself. File names, code divisions, library and sublibrary names, code structure, they make it all easier to understand, not just the semantics but the intent.

When you re-read what you've done, you want to make an easy mental list of the changes, that optimizations the documentations, all make it easier to understand and what needs to be improved.

None of this is code-related. Your brain ignores the things it understands, and when you've looked at over 300,000 lines of code (LOC) you ignore the mundane declarations, the clear logics, everything about what is in front of you. If the code is like this, then your effort going forward will be minimal.

Here's another reality: if you ever need to improve speed, or improve memory size for a restricted memory footprint, then you need to know what changes need to be made easily and quickly to fit the new needs. Every change must incorporate code working differently than it was developed. If you haven't wanted for the right packet on the internet, then you haven't dealt with things like interrupts, If you need new user inputs, then you need to setup, calibrate, and devices across many kinds of input/outputs schemas like single bit, multi-char, serial buses, and so on. If you need to divide the work over many processes, you need to know how to setup shared memories, mutexes, and what to separate. Your brilliant code, your intelligent designs, all depend on the sustainability in the long term.

Tom Calloway's Take

Tom Calloway was so frustrated with Chromium this is what he spent time compiling in 2009:

> **How you know your Free or Open Source Software Project is doomed to FAIL (or at least, held back from success)**
>
> This was inspired by my recent efforts to look at Chromium, but these are just some of the red flags I generally have observed over the years written down(and denoted as points of FAIL (POF)).
>
> == Size ==
>
> * The source code is more than 100 MB. [+5 POF]
>
> * If the source code also exceeds 100 MB when it is compressed [+5 POF]
>
> == Source Control ==
>
> * There is no publicly available source control (e.g. cvs, svn, bzr, git) [+10 POF]
>
> * There is publicly available source control, but:
>
> * There is no web viewer for it [+5 POF]
>
> * There is no documentation on how to use it for new users [+5 POF]
>
> * You've written your own source control for this code [+30 POF]

```
* You don't actually use the existing source control [ +50 POF ]
== Building From Source ==
* There is no documentation on how to build from source [ +20 POF ]
* If documentation exists on how to build from source, but it doesn't work [ +10 POF ]
* Your source is configured with a handwritten shell script [ +10 POF ]
* Your source is configured editing flat text config files [ +20 POF]
* Your source is configured by editing code header files manually [ +30 POF ]
* Your source isn't configurable [ +50 POF ]
* Your source builds using something that isn't GNU Make [ +10 POF ]
* Your source only builds with third-party proprietary build tools [ +50 POF ]
* You've written your own build tool for this code [ +100 POF ]
== Bundling ==
* Your source only comes with other code projects that it depends on [ +20 POF ]
* If your source code cannot be built without first building the
bundled code bits [ +10 POF ]
* If you have modified those other bundled code bits [ +40 POF ]
== Libraries ==
* Your code only builds static libraries [ +20 POF ]
* Your code can build shared libraries, but only unversioned ones [ +20 POF ]
* Your source does not try to use system libraries if present [ +20 POF ]
== System Install ==
* Your code tries to install into /opt or /usr/local [ +10 POF ]
* Your code has no "make install" [ +20 POF ]
* Your code doesn't work outside of the source directory [ +30 POF ]
== Code Oddities ==
* Your code uses Windows line breaks ("DOS format" files) [ +5 POF ]
* Your code depends on specific compiler feature functionality [ +20 POF ]
* Your code depends on specific compiler bugs [ +50 POF ]
* Your code depends on Microsoft Visual Anything [ +100 POF ]
== Communication ==
* Your project does not announce releases on a mailing list [ +5 POF ]
* Your project does not have a mailing list [ +10 POF ]
* Your project does not have a bug tracker [ +20 POF ]
* Your project does not have a website [ +50 POF]
* Your project is sourceforge vaporware [ +100 POF ]
== Releases ==
* Your project does not do sanely versioned releases (Major, Minor) [ +10 POF ]
* Your project does not do versioned releases [ +20 POF ]
* Your project does not do releases [ +50 POF ]
* Your project only does releases as attachments in web forum posts [ +100 POF ]
* Your releases are only in .zip format [ +5 POF ]
* Your releases are only in OSX .zip format [ +10 POF ]
* Your releases are only in .rar format [ +20 POF ]
* Your releases are only in .arj format [ +50 POF ]
* Your releases are only in an encapsulation format that you invented. [ +100 POF ]
* Your release does not unpack into a versioned top-level
directory (e.g. glibc-2.4.2/ ) [ +10 POF ]
* Your release does not unpack into a top-level directory (e.g. glibc/ ) [ +25 POF ]
* Your release unpacks into an absurd number of
directories (e.g. home/johndoe/glibc-svn/tarball/glibc/src/) [ +50 POF ]
== History ==
* Your code is a fork of another project [ +10 POF ]
* Your primary developers were not involved with the parent project [ +50 POF ]
* Until open sourcing it, your code was proprietary for:
* 1-2 years [ +10 POF ]
* 3-5 years [ +20 POF ]
* 6-10 years [ +30 POF ]
* 10+ years [ +50 POF ]
== Licensing ==
* Your code does not have per-file licensing [ +10 POF ]
* Your code contains inherent license incompatibilities [ +20 POF ]
* Your code does not have any notice of licensing intent [ +30 POF ]
* Your code doesn't include a copy of the license text [ +50 POF ]
* Your code doesn't have a license [ +100 POF ]
== Documentation ==
* Your code doesn't have a changelog [+10 POF]
* Your code doesn't have any documentation [ +20 POF ]
* Your website doesn't have any documentation [ +30 POF ]
=== FAIL METER ===
0 POF: Perfect! All signs point to success!
```

> 5-25 POF: You're probably doing okay, but you could be better.
> 30-60 POF: Babies cry when your code is downloaded
> 65-90 POF: Kittens die when your code is downloaded
> 95-130 POF: HONK HONK. THE FAILBOAT HAS ARRIVED!
> 135+ POF: So much fail, your code should have its own reality TV show.
> Anyone want to guess how many POF Chromium has?

If your code is built for developers, then remember they have a say in how good your code is. Developers are your employees that hang their hat elsewhere. Developers KNOW ENOUGH to fix minor problems without asking (as I write this, I fixed my WordPress installation of Dennis Hoppe's encyclopedia without help). If your code is for users with NO TECHNICAL ABILITY, then how much worse will they rate it if it doesn't work?

Enough of failure, let's look at the unipolar success of an open source, bazaar-developed, software goliath.

Chapter 7

Linux: The Pinnacle of Software Reuse

All Conditioned Things Are Impermanent
When One Sees The With Wisdom
One Turns Away From Suffering.
—Siddhartha Gautama, The Buddha

Let's explore the world's most successful open source project: linux.

The first linux kernel (operating system) was release in 1991 by then Finnish student Linus Torvalds. The first official version was 1.0 released in March 1994. I was graduating from Royal Military College at that time- readying for final exams, and went on to serve in Army units. So my awareness didn't come around for several decades.

Today is some day in March, 2022. That's 28 years of an open source CPU operating system linux kernel, originally made because corporations made the Unix kernel too expensive for young hackers to use. Do you believe that any of the original code resembles any of the original written files. Some of the file names, probably for continuity. Some of the names of elements of it, sure. But very little of the original code remains. As of this date, the main trunk Linux kernel is on version 5.17-rc5 (release candidate 5). It is maintained and developed by hundreds of thousands of contributors. All this despite the recorded fact that Linus Torvalds is a horrible person to work with. The complaints are legendary and from what I've seen it's probably deserved. As a difficult individual to get along with, there's not been a shortage of turmoil, upheaval, calamities along with all the success.

The kernel is supported by Google, Microsoft, Red Hat/IBM, and many other Fortune 500 technology companies. The book by Eric Scott Raymond, The

Cathedral and the Bazaar (2004) illustrates the consequential intellectual battle between Microsoft, the cathedral of software developing, and the Bazaar, the free, open source Linux formless insurgent forces of Linus and an every varying array of developers. Microsoft, for all it's commercial might, could not out compete with the independent, asynchronous massively parallel subversive and altogether chaotic development in Linux. About the middle of 2010, the corporate titans sued for peace with Linux and worked to exploit it, not stop it.

Let's not expect, after heeding the words within this humble book, I will propel you to the near limitless level of this computer colossus.

Google claims to have a code base of greater than 2 billion lines of code. There is no way that anyone, any group, will understand all of this. From a sub, sub, sub, project in this repository (BTW I've read Google's former CEO's Eric Schmidt's open source project lex back from before he got rich and famous and it's, frankly, pathetic) will undergo lobotomies language changes, QA testing regimes, regression testing, uprevisions, and forks and so on and live many lifetimes.

But if you can make your code compelling, outlast the first set of developers to hang on a decade, you have made something significant. A project that stands the test of time is a project worthy of merit.

Various nomenclatures, like dialects spoken by an in-crowd of technocrats, perhaps adopted from various education and company settings, make it hard to unify along a common set of ideas that will map one to one with ideas in anyone's head.

The most hated design decision I have seen inside many code bases, is the refusal to make a headless version of the software - without a GUI - available at the same time as the project. It's a common way to obsolete all the good ideas inside your code. Most of the GUI kinds of the past 20 years were poorly thought out, were adopted provincially, and many disappeared as the better designs - ones that were portable across operating systems - appeared and took over. It's an added layer of complexity to try and get a buggy GUI extracted from on top of the members and methods that people really want to use. GUIs from the late 1970's and 1980's are hard to find any source code at all, so there would be a struggle to get these working at all. If you want to help people adopt GUIs on top of your code so be it, I don't think that's a bad thing, but make sure your work can exist without any one GUI. The more things you connect to, the larger testing regime to make it all work, and keep it working as more code is added. If you were smart, you'd let people fork GUI types as they prefer and make your code work with all of them. That way you don't pick the losing horse, you don't alienate people that want the other work alone, and you give yourself things like science fair projects (more on this later) for added co-investment. This is a great way to get companies to invest in your work, let them manage their own GUIs. But maintain a headless version of your system so it can work anywhere, from a server to a smartphone.

Back to the question about does your code look the same? There are probably hundreds of thousands of code improvement commits (inserted fixes like diff and patch) to the linux kernel. When numbers like that are there are very little line by line changes still like the original main trunk release 1.0. But. if you go to include headers, which must be common for people over generations that never delve into the kernel, and they look remarkably similar to the include headers through previous generations. More, different headers perhaps, but a familiar code partitions, and so on. Devices have increased, by type and commercial brand, but the location of devices relative to the rest are in the same spots. Communications protocols have added for the latest wire communications, more protocols have been added, but they still use the same file paradigm for most actions to open, close, read and write as they did in the original 1.0 version.

Making a successful reuse library / repository is a commitment to the long term.

More than great code.

More than the best people.

More than good fortune (because you can expect all kinds of fortune).

What this book sets out it the way to format the best outcomes from my learned and experience reading code projects that went extinct.

Like Michael Kerrisk's man7.org project, in addition to his great book The Linux Programming Interface, man7 provides easy, well-laid exemplifying of all the powerful capabilities within the linux kernel, my code project example, RULA - Re Use Library Abstraction, founded on the IEEE SA Open software gitlab zone is a starting point. Unlike Mike's work, where (I can confirm that Mike is a great guy after interacting with me over the COVID lockdown he graciously forwarded me a pdf copy of his book when mine was stuck in my lab on the army base.) Mike's work is offered as is to be taken as is and used the way expected within the Linux kernel, my RULA library is meant as a starting point for your own work, where you can expand, chop, reorder and so on as you like. Go ahead and clone your own main trunk and let me know occasionally if you are still there. Expand in any way you wish, I'd love to hear back about your insightful work.

adherent (noun) a person that has membership in some group, association, or religion.

It's interesting to note that you could make the mapping between someone's allegiance level as a function of their emotional attachment to a group. As an outsider, one has tepid feelings. As an association member, feelings about the group are in the middle of the range of social attachment. As a member of a religion, the emotional attachment is at the peak of the human experience. What's more important to your software success - your *lines of code* or *number of followers*? Your

Why did I feel my observations were valuable enough to write them in a book?

Because I've spent countless months, helplessly, and hopelessly scanning many projects for my own code needs. I've scanned code from famous people like Ken Silverman, the young C prodigy that sold the original game engine for DooM to id software, to university groups of professors and students making community projects like Player/Stage/Gazebo, funded by DARPA to accelerate progress in mobile robotics, to dreamers and schemers and vaporware make believers. Many projects start full of enthusiasm. If you think that writing code is tough, and it shouldn't be if you use all those design methods they teach you, fill your head full of, at engineering colleges, then imagine how hard it is to arrive at a tarball, unpack it, run doxygen, or just reading the code directly or the directory tree, and not having those designs, not having much more than ReadMe.txt file I have downloaded over 5000 projects in my career.

Most times I wanted to leverage someone's work positively so I could speed myself to results. The outcome was a career full of lessons on how to avoid my predicaments.

Whatever you believe in your code, if is unclear, if it is laid out nonlinearly, and if it is to incorporate various parts contracted out into a longer term rerole-back into the project as new messages, new processes, new threads, new services that are complimentary then organization and form become bigger and bigger in importance than code generated daily. I am a silent witness to many individual efforts that went for naught.

The biggest problem with any project is understanding.

All the human errors, great and small, that we all suffer from on a transient or periodic basis, arrive at from misunderstanding. So, the solution here is that we all must make things more understandable, greater understanding and lower confusion, and that will allow greater applicability, more ease of use leads to more use, more ways to use it will lead to more demanding absolutes from anyone and anything, will guide more people to your work to selfishly satisfy their needs easily, and cheaply and without much fanfare nor to-do. The complimentary goals. ease of use, understandably recognition and project can take an unknown trek and take and can be weighed down by many global factors you cannot control and may have unforeseen global events like war, political upheaval, that will ruin your best-laid marketing plans you pay for.

Chapter 8

Organizational Considerations

Complexity Expands with Knowledge Hierarchy

The second biggest problem that fails software projects is the problem of hierarchies. The layers of things make it harder to see all that's inside. The layers, the kinds of layers that absorb content, agglomerate ideas, and stratify knowledge make the needed important details hide out of obvious sights. It is both a blessing and a curse.

But, how can one man or even a small team of developers write all those diverse ways for people to understand quickly and efficiently? That brings me to another project area dutifully funded by the USA government and was fully intended to make this problem manageable.

Another effort, outside the scope of software in the field of epistemology (of course few like to look across the borders of one research area as they all tend to get comfortable in their sameness) brought about a mechanism for knowledge transportation, exchange, and storage. It's funny, you pay people to think apart, but left to their own devices they make it easier to cooperate and refuse to consider their own principles (principles: strong beliefs) to be where the research must examine. This other field was remarkably and gifted in examining the language of knowledge using the seminal work of Charles Sanders Pierce. There is a gold mine in the works of Charles Sanders Pierce, if only we delved as deeply there as we did in foreign entanglements. Another American mathematician stands out as one that guides the efforts towards a standard for knowledge base acquisition, transportation, and storage. John F. Sowa wrote extensively in the construction of conceptual graphs and the main architect of the ISO/IEC 247707:2007[1][1] Common Logic Interchange Format and Conceptual Graph

[1] https://msb.isolutions.iso.org/obp/ui#iso:std:iso-iec:24707:ed-1:v1:en

Interchange Format that pins a mechanical/mathematical process to the processing of information. This is a needed background effort to the foreground operation of making library functions into an understandable set of concepts for many different humans to identify. The US Department of Defence (DoD) invested in the making of a knowledge association system that went into projects like Central Archive for Reusable Defense Software (CARDS), STARS, and grand hierarchies like DoDAF

Just to give you the problem of human mapping of all data that might be used by anyone person to find things, consider these simple - oversimple - examples just surrounding the use of acronyms, mathematical categories and keywords as things that point at one thing people try to find. Think of them all as occupying the same zero level space, and anyone of them can point to the object someone is looking for. In this case, there is a cardinality of one. Define the things you are searching for are set x elements of A and the total keywords are set y as elements B:

Stratifications

stratification (countable and uncountable) the process leading to the formation or deposition of layers.

If there is one thing a lead developer, business manager, system architect, executive vice president can influence, if not every line of code from inception to adolescence, is the stratification of the project. It's one of the first factors that starts as a cliché slide for investors and management meetings, that then gets tossed over the wall to the code monkeys to forget and tarnish, whichever comes first. Management must NEVER lose control of the software stratification. Here's why:

Stratifications make details nonobvious:

Layers by organization.
Layers by categorization.
Layers by association.
Layers by development.
Layers on alpha/numeric ordering.
Layers on temporal ordering.
Layers on reverse temporal ordering.
Layers on personal preferences.

Erickson's Rule: the greater the stratification, the lesser the information is easily accessible.

From Erickson's Rule, we realize that information hiding and layering are the causes of difficulty with software. Ordering, the function that makes stratifications, gives the many ways to confuse the point for those that didn't make them. We build layers into things like we store papers in file folders in filing cabinets. We want to ignore the details, summarize what we did previously, and only pull open the file folder when we need it. It's both an advantage and a curse. On your side, your stratifications make sense. When you open someone else's filing cabinet, the order makes no sense, to you. Software is and will remain a distillation of human thought, the less you understand where and why those thoughts are, the less control you have over the finished product. You need your software to make sense to everyone involved in the process. For those that claim a project failed because the software didn't work, I state the project masters lost control of the stratification so they couldn't impact the parts that weren't working well enough.

If you don't understand where people are putting things, then how can you honestly report on progress and quality?

Stratification is inevitable, with the failings of any other stratifications of layering, even in software, knowledge, and every other field including epistemology. We can't expect humans to benefit from your work until you pave the way for them.

The problem of understanding is larger and wider than the software domain. These are the problems of epistemology so it's been around over 2000 years. Plato, Socrates, Pythagoras and others didn't solve it then. The knowledge of problems is the problem of knowledge, they are one and the same. Every person will need their own way to find their way to understanding WITHIN YOUR WORK. Since you don't expect to sit on the shoulder of every

Areas to Improve without total renovation:

1. Globalize variables.
2. Add localization.
3. Language conversion of code and documentation.
4. Isolate security elements.
5. Reformat error codes.
6. Replace local printf() type functions with an operating system version like perror() and associated functions and structs in Linux.
7. Customize code for an operating system in general (OS, Proper model, Dialect, threaded operation, etc.).

8. Lobotomize: cut off the head GUI.

9. Functorify - Replace the function model with another one without replacing the data structs and members. For example, instead of get/set for system variables you replace with network confirmation versions like Transact with a confirmation and a data mutex.

10. Employ common design patterns in place of custom code.

The critical path in the age of distributed, asynchronous, software development.

The old standard we all learned back in the olden days was based on an understanding that teams were stood up and given a full design cycle, perhaps years, in order to chart all the work out from a complete life cycle with a waterfall model. Then came the acceptance that the best way was to adopt a spiral model - which the unix development model (make a proof of concept then improve it) is based on, was a way to rapidly approach the target without as much lead time, perhaps accepting the reality that people join work done by others.

What this means in the latest software is rather than a single waterfall or a first iteration spiral, you have to predict and anticipate a multi-pronged attack. You need to prepare for many paths open and available at the beginning. You will be better served if you get ready for many critical paths as alternatives. It gives you options to community fill development, to contract out development, and to test at alternate milestones that lead you to a faster success state.

With that reality of more and perhaps better, you just need to concentrate on three keys:

1. **Make it clear**, make it understandable by potential adherents.

2. **Make it yours**, commit to do all the background tasks like git documentation and release versions as much as the code. Chase down bugs.

3. **Divide and conquer**, make it clear what segments are and be flexible and accepting of a potential multitude of non-critical paths to become the critical path making them all live. The more flexible you are (and as Linus Torvalds is accused of, promiscuous) then you aren't working your way into a dead end as much as charting many paths to victory.

One of the bright spots I can report to you concerns the ideas of microservices. A microservice is a smaller (presumably over internet messaging systems) middleware (a transport medium for many other kinds of applications and executables) concept is that if you are aware of it, then you can avoid a lot of the headaches and dead ends by designing to make a smaller, less ambitious but no less effective microservice instead of a large monolithic program. It give you a way to pay for features and extra services from other providers by dividing the work. It makes the effort smaller to start but can expand in any direction you wish. It can give you a marketing copy in initial stages for venture capital security or product pre-sales, and then even off ramps if the situation for profitability is downgraded. It even gives you a way to cancel future work based on test and evaluation of submitted code. It allows you to target effort at flaws, defects, and overall performance in any direction.

There's a good book on microservices from Sam Newman, called, Building Microservices that goes over the topic from a macro level and presents new design patterns to suit the workings. Here's why microservices is a good concept to use; for most of the reuse models presented in this book one starts from working code to better working code. For microservices you can simply plan in upgrades with new code and deprecate the old code you started with in the process. To the user, no drastic changes can be detected.

Tools Identification

For each activity in the existing or potential reuse process inside your organization, you can start at reuse by assessing your own tools. Get your team to describe how your existing tools might support reuse- perhaps because they process source code/documentation in a novel and perhaps proprietary way. Assess each potential tool's importance to reuse - do they provide your organization with a unique advantage? Assess the feasibility of existing tools - will retooling or reconfiguring these tools make a significant improvement in software reuse feasibility. Would automated tools drastically accelerate your reuse activity? Do you possess tools to assess the maturity development level of bespoke software?

Hire the Coder

There's a simple change one can make to the entire mindset of organizing for a new goal of software reuse. The simplest way to get the valuable asset of code modifications right without a lot of fuss, organization upheaval, tools learning and so on, is to consider hiring the coder and his code as a mercenary for the duration of the intended work. The people closest to the source are the most knowledgeable, if it hasn't been too many years. People put a lot of effort into their own code bases, and then work a job to support themselves. Or it's a hobby, and so on. The way you hire them is up to you, whether it's just to document the code, make the changes you need, or as a full time employee designated to maintain the merged code forever. It’s just another way to handle the work by considering the people involved as committed to your success as they are to the success of their hard work, paid or not.

Chapter 9

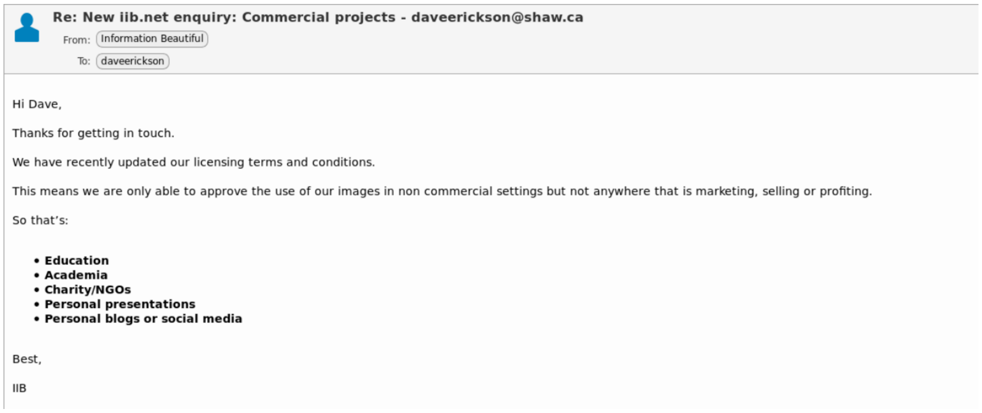
Legal Aspects of Code

None of this forward-looking interpretation expounded herein can be assumed to apply directly to any singular element of code that you may decide to leverage. One must understand the ways to reuse under fair use. Here's a beginning to a very large area on it's own.

Rejection Is Part of The Game

In writing this book, I had to come across many idea sources for various aspects of this book. In the spirit of reuse, I wanted to leverage previous work in demonstration of leadership, and give credit where credit is due so I needed to ask for permission to use more than just a little snippet so the original authors and artists might be heard in their own voices and not through my interpretation. After all, that's what you are doing when you use other's software, you are reapplying another's ideas. Of course, this won't be possible for everyone, you will find people that refuse to share for any number of reasons. And all reasons are valid, to a point. Rejection is part of the game (Figure 9.1). I applied to David McCandless to use his very large and symbolic lines of code graph about the relative size of various "code bases"[1] up to the human genome at (his count) 3,300 billion "lines of code". I wanted to be fair and use a good exemplar for the benefit of my audience but also make you aware of his work. This is a snippet (Figure 9.2) of his image included to demonstrate to my audience through my criticism of his actions exactly what people don't understand about the law of fair use. Expect to get rejected in your requests to use. But that doesn't mean you are in the wrong if you use them with fair use principles in mind.

[1] David included the human DNA which isn't strictly computer operation code - it's human "building code".



Re: New iib.net enquiry: Commercial projects - daveerickson@shaw.ca

From: Information Beautiful

To: daveerickson

Hi Dave,

Thanks for getting in touch.

We have recently updated our licensing terms and conditions.

This means we are only able to approve the use of our images in non commercial settings but not anywhere that is marketing, selling or profiting.

So that's:

- **Education**
- **Academia**
- **Charity/NGOs**
- **Personal presentations**
- **Personal blogs or social media**

Best,

IIB

Rejection is part of the game.

Rejection doesn't mean you don't have a reasonable grounds to use their work, or a part of their work, under fair use. And that's what he just did; he furnished me a perfect example of fair use.

What I wanted to do was share his ideas, unvarnished and untainted by my analysis or need to criticize[2]. I believe this is in the spirit of my book. I wanted his work to stand on it's own. But in the end, by rejecting my request for permission to use his copyright what he has done, in my analysis, is to furnish me with a perfect exemplar of how and why copyright owner's are in serious ignorance of what fair use means and how any copyright is not an absolute. That's the purpose of this chapter identically - Legal Aspects of Code.

Think about it; no one jumps out of a car and hits you with a Cease and Desist letter when you say, "Mickey Mouse". Disney wants you to chat about your time at Disney Epcot Centre. What they can't do is stop you from in the same breath explain, "That Disney's Epcot Centre is old and crumbling." Because that's your right under free use to criticize.

The Four Considerations of Fair Use:

Back to my target. In order to demonstrate that you can still use something provided you maintain the spirit and letter of the law of fair use as I will lay out here, my considered fair use[3] is presented by clause and term of the "US Fair Use Explanation of Section 107" of The Copyright Act. These elements would

[2] I never set out to castigate David McCandless nor his work, but when he furnished me with what he did, the eureka light bulb went off about how to leverage what was on offer.

[3] More information on Fair Use: www.copyright.gov

the core of any case judged for or against you, so you must find your way through them in the beginning. Read the fair use Section in the country you live in before you use. You must consider the following in the US:

First consideration (these considerations are copied *verbatim* from the US Government website More information on Fair Use: www.copyright.gov[4]):

> "Purpose and character of the use, including whether the use is of a commercial nature or is for nonprofit educational purposes: Courts look at how the party claiming fair use is using the copyrighted work, and are more likely to find that nonprofit educational and noncommercial uses are fair. This does not mean, however, that all nonprofit education and noncommercial uses are fair and all commercial uses are not fair; instead, courts will balance the purpose and character of the use against the other factors below. Additionally, "transformative" uses are more likely to be considered fair. Transformative uses are those that add something new, with a further purpose or different character, and do not substitute for the original use of the work."

For my work, I am creating a commercial use work, and that may be considered a violation of fair use as detailed above in "Purpose and Character of the use". However, what I am doing is not simply copying the information. I am adding a transformative character to the introduction of the work to describe how fair use may apply. I am showing both a request for use and my criticism of that rejection as part of the analysis of the data, which by the way isn't David McCandless's data, it is public knowledge, his graph is merely a transcription of public data. I need to show my audience what the fuss was over, his images are helpful to some. The lack of his image would dampen the understanding of the relative size of lines of code code bases in consideration. Under the first consideration, my use of David McCandless's work is inevitably fair use.

Second consideration:

> "Nature of the copyrighted work: This factor analyzes the degree to which the work that was used relates to copyright's purpose of encouraging creative expression. Thus, using a more creative or imaginative work (such as a novel, movie, or song) is less likely to support a claim of a fair use than using a factual work (such as a technical article or news item). In addition, use of an unpublished work is less likely to be considered fair."

[4] Copyright US Government 2022.

I am using a partial image from a previously published book, not from his secret collection, not from an upcoming rehash of the same book. I obtained the image from the internet not his book, from a third party already critiquing his work. As mentioned above, David McCandless took data from public sources about the relative lines of code involved in various organization's work, Microsoft's operating system Windows 2000 for example, and put that public information in the same chart with words, numbers, and colours. Microsoft own's information through copyright on Windows 2000 about the size of the code base is not David McCandless'. So David McCandless is using other's copyright information via - you guessed it - fair use. The nature of the chart is factual work quoting the technical specification of the numbers of lines of code. Do you believe David McCandless sought a copy of the software and counted the lines of code by himself? No, he gathered the data from the actual source (Microsoft) or via a third party that had that information.

He admits this himself in the book description (from Information is Beautiful):

> *"Facts, statistics, issues, theories, relationships, numbers, words - there is just too much information in the world. We need a brand new way to take it all in. 'Information is Beautiful' transforms the ideas surrounding and swamping us into graphs and maps that anyone can follow at a single glance."*

He takes data from other sources, and computes numbers and presents them in more than one color. That's it. Under the second consideration, my work is fair use.

Third consideration:

> "Amount and substantiality of the portion used in relation to the copyrighted work as a whole: Under this factor, courts look at both the quantity and quality of the copyrighted material that was used. If the use includes a large portion of the copyrighted work, fair use is less likely to be found; if the use employs only a small amount of copyrighted material, fair use is more likely. That said, some courts have found use of an entire work to be fair under certain circumstances. And in other contexts, using even a small amount of a copyrighted work was determined not to be fair because the selection was an important part—or the "heart"—of the work."[5]

[5] Here is the footnote to the webpage, presented: "In addition to the above, other factors may also be considered by a court in weighing a fair use question, depending upon the circumstances. Courts evaluate fair use claims on a case-by-case basis, and the outcome of any

Under the third consideration (based on the Amazon book URL webpage since I have never read nor bought his book[6] and never will after this), which I must use third party information of this critique of his book, that David's work is 255 pages. In my criticism of David's work, I am using less than one-half of one page of 255. In percentage terms, that's 0.2% of the total work.

In actual fact, one of the commenters points (on Amazon) out that David McCandless is in fact stealing from his own work from 2009. His 2012 book is a resteal of The Visual Miscellaneum: A Colorful Guide to the World's Most Consequential Trivia The Visual Miscellaneum: A Colorful Guide to the World's Most Consequential Trivia. Egads, he's stolen from himself. Someone, call a lawyer!

Fourth consideration:

> "Effect of the use upon the potential market for or value of the copyrighted work: Here, courts review whether, and to what extent, the unlicensed use harms the existing or future market for the copyright owner's original work. In assessing this factor, courts consider whether the use is hurting the current market for the original work (for example, by displacing sales of the original) and/or whether the use could cause substantial harm if it were to become widespread."

My criticism of David McCandless's rejection has not said a derogatory word about his work. My work is about software reuse, not pictograms (David is in fact copying the idea of pictograms - see definition below - made by others). I am not in the data visualization business, and this work isn't a direct nor indirect competitor of his books. In fact, David is copying the pre-made style of infographic (see definition below) which he neither invented nor copyrighted. Probably because infographics are in the public domain. A complaint by original authors of infographics (see definition below) books might be better made against David McCandless for his 2012 book.

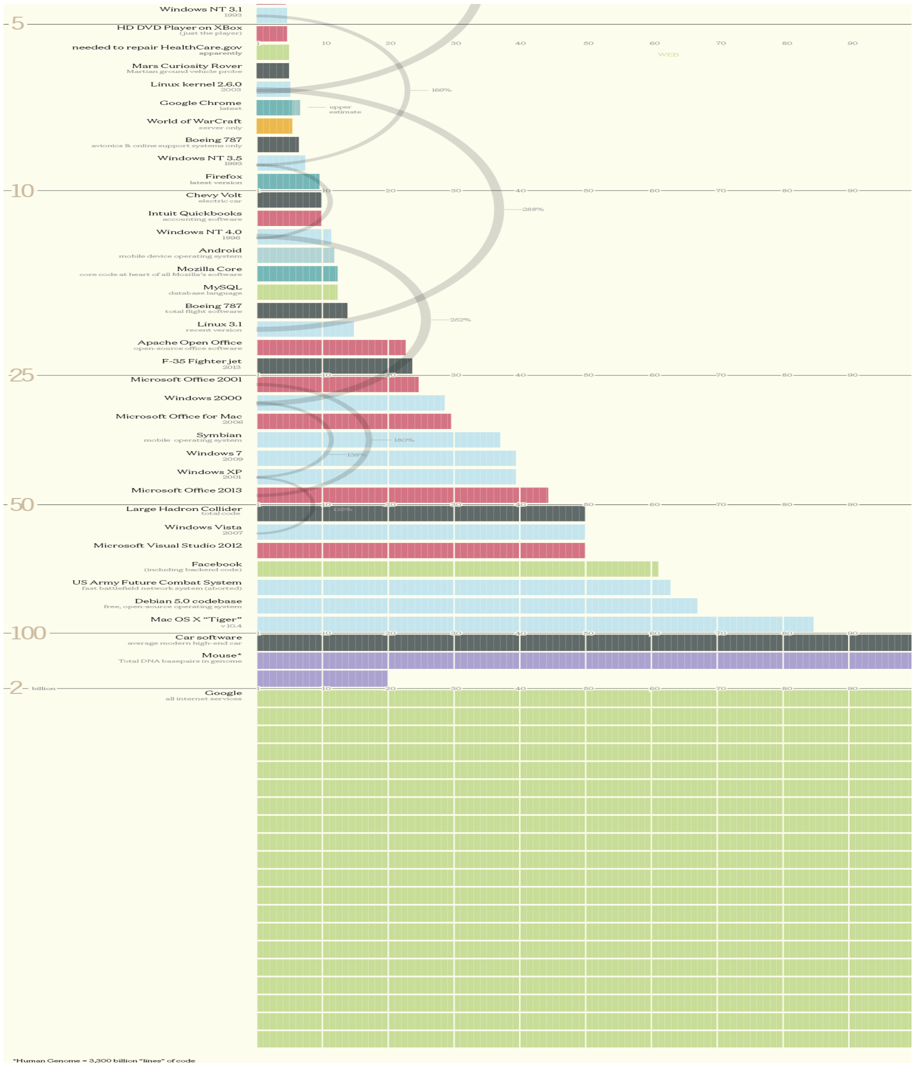
pictogram A picture that represents a word or an idea by illustration.

infographic A visual representation of information.

given case depends on a fact-specific inquiry. This means that there is no formula to ensure that a predetermined percentage or amount of a work—or specific number of words, lines, pages, copies—may be used without permission."

Please note that the Copyright Office is unable to provide specific legal advice to individual members of the public about questions of fair use. See 37 C.F.R. 201.2(a)(3). Copyright US Government 2022.

[6] Information is Beautiful



David McCandless's infograph (partial - included through fair use for analysis as described in Chapter 9)

Here's my analysis of David's rejection, in his matter of fact and completely non-considered way, is that the only person to harm his long term prospects for sales is David McCandless and his team. There's a saying, "no news is bad news". In the modern deluge of data that we are all inundated with, it's always good press to get press for free. More reference means more awareness. I had no idea his work existed until one DuckDuckGo web search. Positive behaviour makes people more cooperative and aware of your work when you ACT magnanimous. To not behave in that manner is a long time loser for yourself. In fact, I am affording David's work a few pages in my work with all the free press that presents. He is most welcome.

You can apply the same *four considerations of fair use* to any potential software reuse. Of course, once you analyze them, pass them by your lawyer for approval.

With my fair use teardown complete, let me end by stating that we are all working for the future prosperity, and while people may reject cooperating with you, it's not the end of story if you have a valid justification to use something, as I did here.

And to show you that there are both commercial and cooperative people out there, here's my permission acceptance from Jacob Beningo from https://www.beningo.com/ (Figure 9.3) for the use of his printf time response in Chapter 12 in a figure. He's quite happy to have some free press. And he wants to be treated with respect, but that's only fair, and what is afforded to every work placed here to illustrate. I include both authors by name and books by reference in this book. I apply copyright notices as directed. I want to lead and demonstrate cooperation for the future prosperity.

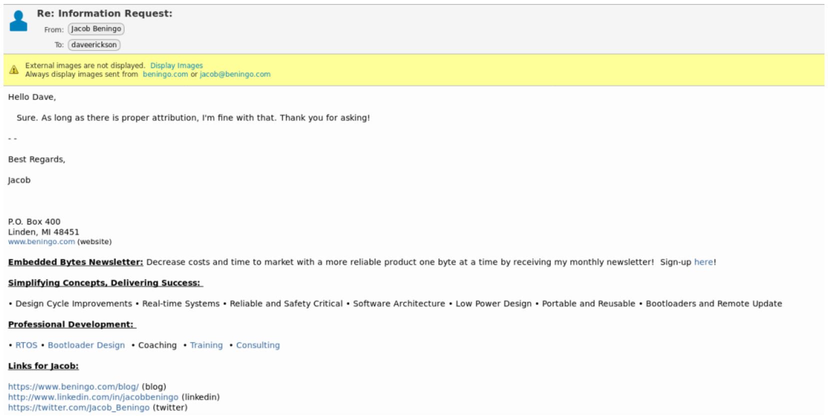
My response to Jacob merely asked him what format and copyright year he wanted. Most people place a copyright direction with their work, explaining how to refer to it. I will use whatever format I am told, or I make one up and send it by email to get confirmation.

By the same spirit, if and when anyone wants to ask for my permission I will always grant it. Why? Because I don't want to be the target of valid negative criticism, as above!

Prevailing Software Licences

As I was wading through some new NASA code, I realized they have updated the licences for some of the software I use. The new core Flight Executive (cFS) uses the Apache (née A Patch EE) 2.0 Licence,

Apache 2.0 Licence is here. There is no way to write a static book about where the organizations exist on the internet, and rapidly change the available choices. Thankfully, this is a multiyear work that will attempt to accommodate.



Re: Information Request:

From: Jacob Beningo

To: daveerickson

External images are not displayed. Display Images
Always display images sent from beningo.com or jacob@beningo.com

Hello Dave,

Sure. As long as there is proper attribution, I'm fine with that. Thank you for asking!

- -

Best Regards,

Jacob

P.O. Box 400
Linden, MI 48451
www.beningo.com (website)

Embedded Bytes Newsletter: Decrease costs and time to market with a more reliable product one byte at a time by receiving my monthly newsletter! Sign-up here!

Simplifying Concepts, Delivering Success:

• Design Cycle Improvements • Real-time Systems • Reliable and Safety Critical • Software Architecture • Low Power Design • Portable and Reusable • Bootloaders and Remote Update

Professional Development:

• RTOS • Bootloader Design • Coaching • Training • Consulting

Links for Jacob:

https://www.beningo.com/blog/ (blog)
http://www.linkedin.com/in/jacobbeningo (linkedin)
https://twitter.com/Jacob_Beningo (twitter)

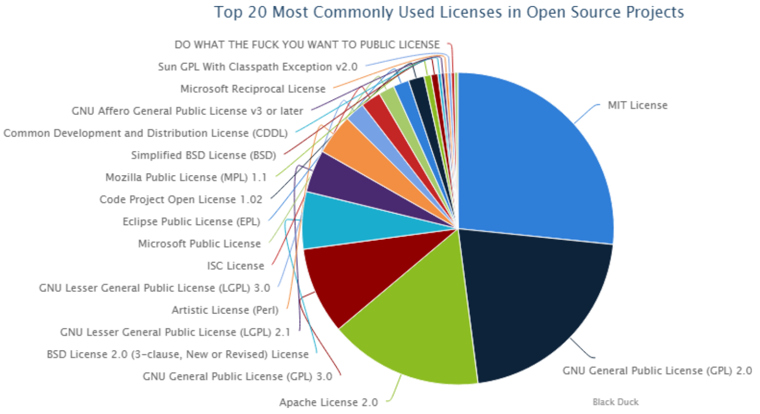
Approved Permission to use copyright by Jacob Beningo.

Licence Organization	Hyperlink
The Apache Software Foundation	www.apache.org Apache 2.0 Licence LICENSE-1.1 LICENSE-1.0
MIT Licence	MIT Licence
GNU Free Software Foundation	GPL- Gnu Public Licence 1.0
BSD Licence	BSD Original Licence
Eclipse Public License - v 1.0	Eclipse Public License - v 1.0

Table 9.1: *According to BlackDuck Software, the commonest public licence distribution back in 2016.*

As you consider software reuse, it's a good idea for managers to become familiar with the flavours of licence out there, and they are all now volumes on their own so it's impractical to add them all verbatim. Nonetheless, I can point you at the key few you will hear of again and again. So reading some of these on your own will alert you to the important rights granted and how they might impact your code. The Apache 2.0 Licence is an irrevocable any use right that basically allows you to do what you wish, the MIT Licence and the GNU Free software Lesser GPL is in the same spirit. The others are more restrictive so you should be aware.

Here are the prevailing licences you will come across:



According to BlackDuck Software, the commonest public licence distribution back in 2016.

Rank	License	%
1	MIT License	26%
2	GNU General Public License (GPL) 2.0	21%
3	Apache License 2.0	16%
4	GNU General Public License (GPL) 3.0	9%
5	BSD License 2.0 (3-clause, New or Revised) License	6%
6	GNU Lesser General Public License (LGPL) 2.1	4%
7	Artistic License (Perl)	4%
8	GNU Lesser General Public License (LGPL) 3.0	2%
9	ISC License	2%
10	Microsoft Public License	2%
11	Eclipse Public License (EPL)	2%
12	Code Project Open License 1.02	1%
13	Mozilla Public License (MPL) 1.1	< 1%
14	Simplified BSD License (BSD)	< 1%
15	Common Development and Distribution License (CDDL)	< 1%
16	GNU Affero General Public License v3 or later < 1%	
17	Microsoft Reciprocal License	< 1%
18	Sun GPL With Classpath Exception v2.0	< 1%
19	DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE	< 1%
20	CDDL-1.1	< 1%

Table 9.2: *According to BlackDuck Software, the estimated public licence distribution is back in 2016. BlackDuck Software Source Licences*

A Software Licence in Full:

Here is the MIT Licence, which I feel is a good starting point to broach the ideas that form a licence. I know most of my work is slanted towards Free Software Foundation, but I exploit what makes the most sense. Which is the heart of all knowledge reuse.

MIT Licence:

> *The MIT License (MIT)*
>
> *Copyright © 2022 <copyright holders>*
>
> *Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:*
>
> *The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.*
>
> *THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.*

Important Points in any Licence

First, the definitions of term and condition:

condition[7] n. a term or requirement stated in a contract, which must be met for the other party to have the duty to fulfill his/her obligations.

term[8] n. 1) in contracts or leases, a period of time, such as five years, in which a contract or lease is in force. 2) in contracts, a specified condition or proviso. 3) a period for which a court sits or a legislature is in session. 4) a word or phrase for something, as "tenancy" is one term for "occupancy."

A term is an objective measure applied to a work (my definition) and written down in a contract, in this case a copyright licence. Of course, I could expand the definition of “measure” to create a full syllogism of all the ideas that are similar if not identical to term and condition, ...*reductio ad absurdum*. I offer that a term is “a measure” and a condition is “a measure that can be measured by others”. So, a contractee lays out concepts inside a legal document that apply measures to conditions that can then be judged by third parties to determine if the actions taken upon a work or with a work or by a work meet the original spirit and letter (intent) of the contractee’s wishes. Clear as mud?

Here are the important conditions and terms - outlined blow by blow - to consider as a manager charting out if there is value in exploiting a specific software source code with a defined copyright licence.

Copy

copy[9] A copy is a true transcript of an original writing.

Can you copy it? Can you change the form to an identical version. This condition needs to be in the document for your mutual interest in sending products out the door with copies of the software running, or server farms operating in the background.

Modify

modify[10] vb (mainly tr) , -fies, -fying or -fied 1. to change the structure, character, intent, etc, of 2. to make less extreme or uncompromising: to modify a demand. 3. (Grammar) grammar (of a word or group of words) to bear the relation of modifier to (another word or group of words) 4. (Linguistics) linguistics to change (a vowel) by umlaut 5. (intr) to be or become modified.

Modification is required by software developers along many dimensions: making things smaller, faster, larger, bug-free, and so on. Code that you can copy but not modify in most cases isn’t desirable. There are two areas where this might not be a problem: data compression (like the ziff compressor known formally as the Newman-Ziff Algorithm[11]) and data encryption (like the SHA-256 data encryption algorithm[12]). That applies to things like video compression and audio compression also. These aren't the kind of things you want to fiddle with unless you want to introduce errors or make things not work as well as the originals.

Merge

merge[13] (mɜːdʒ) vb 1. to meet and join or cause to meet and join 2. to blend or cause to blend; fuse.

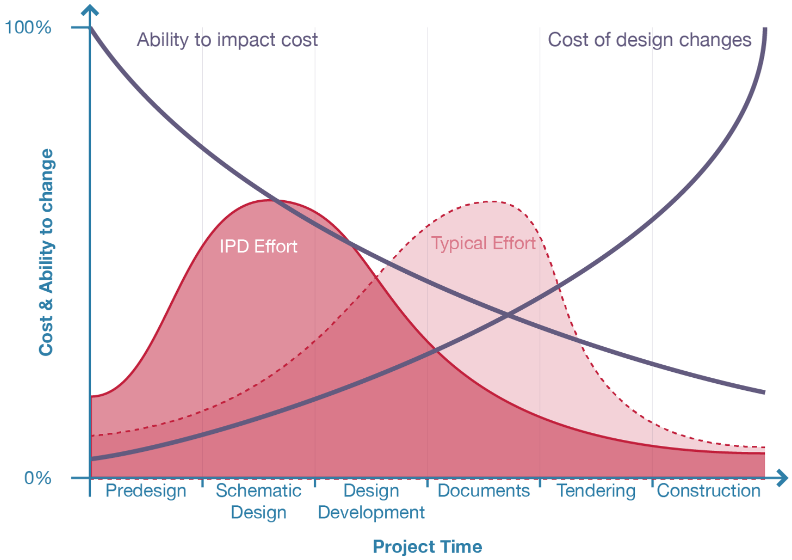
The ability to combine one work with another applies in hardware, firmware, and software. One must be able to bring them into each other's proximity. This won't be a problem if you intend to use one codework on one machine and then network communicate with it over, say the internet, to another machine using different code. One example I use in this book is

Integrated Project Delivery and Multi-Party Agreements

If software companies have been around for many decades and reuse still hasn't been institutionalized, then there must a new structure to handle the novel needs of business to make all of this practicable. One way is to expand the horizons of the business arrangement to meet the additional needs of reuse explicitly. The way I can recommend comes by way of the North Sea oil and gas companies and how to bring in many companies in a deal that handle all the risks and complexities of the work: Integrated Project Delivery (IPD). Here's how the American Industry of Architects describes in AIA's 2007 White Paper: Integrated Project Delivery: A Guide(pg.2) described Integrated Project Delivery teams:

> Integrated Project Delivery (IPD) is a project delivery approach that integrates people, systems, business structures and practices into a process that collaboratively harnesses the talents and insights of all participants to optimize project results, increase value to the owner, reduce waste, and maximize efficiency through all phases of design, fabrication, and construction. IPD principles can be applied to a variety of contractual arrangements and IPD teams can include members well beyond the basic triad of owner, architect, and contractor. In all cases, integrated projects are uniquely distinguished by highly effective collaboration among the owner, the prime

[11] https://pypercolate.readthedocs.io/en/stable/newman-ziff.html
[12] https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm



The IPD Cost Value Proposition for integrating software reuse into regular design-maintenance operations: This is the value to untrap as soon as possible.

> designer, and the prime constructor, commencing at early design and continuing through to project handover.

That's a lot of corporate psycho-speak for a simpler explanation: bring in different kinds of teams to the project, invite them to help formulate your design plan, and allow these extra hands a fixed cost and potential for added profit to help you maximize your code. It's that simple.

IPD in software reuse means bring in two different kinds of teams: salvage software crews that can search and find parts you need ahead of your design team, then educate your team to use them faster than designing themselves. On top of salvage crews, find another or same team to focus on reuse of your software as you make it to be ready to deliver the back end again for different customers. That's potentially three sets of eyes looking at your software requirements in three time directions: from the past to the present (salvage team), from the present to the future (your design team), and the future to the horizon (software reuse for other customers). That's a newfound way to make it all happen.

Hire one team, hire two code mercenaries. That's IPD in my vision.

Here's a table from Integrated Project Delivery: A Guide describing the project commonalities:

Traditional Project Delivery	Commonalities	Integrated Project Delivery
Fragmented, assembled on “just-as- needed” or “minimum-necessary” basis, strongly hierarchical, controlled.	team	An integrated team entity composed key project stakeholders, assembled early in the process, open, collaborative
Linear, distinct, segregated; knowledge gathered “just-as-needed”; information hoarded; silos of knowledge and expertise	process	Concurrent and multi-level; early contributions of knowledge and expertise; information openly shared; stakeholder trust and respect
Individually managed, transferred to the greatest extent possible	risk	Collectively managed, appropriately shared
Individually pursued; minimum effort for maximum return; (usually) first- cost based	compensation/ reward	Team success tied to project success; value-based
Paper-based, 2 dimensional; analog	communications/ technology	Digitally based, virtual; Building Information Modeling (3, 4 and 5 dimensional)
Encourage unilateral effort; allocate and transfer risk; no sharing	agreements	Encourage, foster, promote and support multi-lateral open sharing and collaboration; risk sharing

Table 9.3: *Project Commonalities: Traditional versus IPD*

In Figure 9.5, the goal of integrated product development is clear. You layer new set of eyes looking at code earlier for many purposes, and gain value faster for wider distribution. Bringing in other specialists earlier avoids the pitfalls of later design changes, and can finish faster by bringing development forward to the left.

Salvage and Fair Use

Adding the capabilities of software reuse companies to your project plans gives your organization a tremendous leap forward into capabilities. With the introduction of a wider software team like an IPD above, you can see a reasonable way to extract as much value earlier on that IPD aims to address.

The sea of software abandoned is vast. According to Pixalate, there are ~1.5 million abandoned apps on Google Play and Apple App Store:

> **Research Finds Over 1.5 Million "Abandoned" Mobile Apps**
>
> Pixalate claims they crawled the App Store and Play Store to analyze all apps available for download based on their last update to determine their degree of "abandonment". Abandoned apps were defined by Pixalate as those apps that had not received an update in over two years, with "super-abandoned apps" having not received an update in at least five years.
>
> Based on the previous definitions, Pixalate found over 650k iOS apps and about 870k Android apps to qualify as abandoned apps. Of those, just about 180k iOS apps and 130k Android apps qualify as super-abandoned. Those numbers, amounting to 1.5 million abandoned apps overall, may appear high. But even more striking is the fact that a number of them still receive a significant amount of downloads. Indeed, approximately 16% of apps with 1-10 million downloads and 6% of apps with over 100 million downloads are abandoned apps.
>
> Another interesting figure that can be found in Pixalate report is apps that have received updates in the last 6 months, which could be considered a measure of actively supported apps, amounting to 1.3 million. Pixalate report also shows super-abandoned apps are comparatively more abundant in the App Store than in the Play Store, although this figure is somewhat harder to interpret.
>
> Not surprisingly, games appear to be a top category for abandoned apps in Pixalate report, with over 50k iOS and Android games having not received an update in the last two years. The other two most-abandoned categories are reference and education. It is interesting to observe that

apps belonging to the finance, health, and shopping categories are those that tend to receive updates on a more regular basis.

A sermon on fair use was laid out earlier in the chapter and an in-depth investigation into fair use in Chapter 9, so I won't reframe the ideas of fair use more than to say it's in your best interest to employ professionals that understand how fair use works to any salvage operations to avoid yourself any long term jeopardy for use. Knowing there are many abandoned sources, the best advice is deprioritize using anything from a belligerent source. It can be avoided by many means, too many to list here.

In general, salvage gives you more ways to deliver faster and cheaper than just the traditional design waterfall and the spiral design model combined. You can bring people to work on the downrange goals for a fraction of the price of your team, perhaps making your performance bonus objectives for a fraction of the cost and risk. The reason is salvage. Teams that scavenge can bring components to your satisfaction faster than the normal design steps.

Salvage teams can dovetail and/or piggy back into your design cycle (piggyback and dovetail are defined in 4) when they've passed alpha or beta testing.

At the other end of spectrum, reuse teams can comb your combined code to make into library interfaces to the code, like making your own game engine out of the pieces redesigned from the original code, to be sold separately for other kinds of customers.

Consider if your plan to follow the critical path towards delivery milestones could be improved by hiring a reuse company to scavenge some of your needs in a parallel track. If they extract and get working pieces before your team gets to design, you can simple adopt and adapt. Bringing teams in to salvage and scrounge the internet for you may hit further milestones faster than directed effort can produce.

With new team members forming an IPD partnership, with complementary goals and extra sets of eyes looking at your work, the potential to untrap the most value of your investment is the way forward.

Chapter 10

Re-Use Library Abstraction (RULA)

Since Everything is a Reflection of Our Minds,
Then Everything Can Be Changed By Our Minds
—Siddhartha Gautama, The Buddha

The gitlab library Re-Use Library Abstraction (RULA) can be found here:
https://opensource.ieee.org/daveerickson/reuse-library-abstraction

This project houses the (to be confirmed) IEEE Standards Association Proposed PAR (Project Authorization Request) entitled: Re-Use Library Abstraction (RULA) as a means to accelerate reuse software library adoption by decreasing library learning uptake time. The goal of this work is to implement a Basic Prototype Exemplar that holds all the factors recommended in the book: (TBP - to be promulgated) The first goal is to showcase the easier to use, easier to understand, and easier to improve aspects that will make it win over users through time-tested improvements to the way things are done to this point. This is the work in progress by example and by experimentation. Stop back at that URL from time to time as I add to it and clean things up.

Here are the major improvements that this RULA library brings to you:

1. Identification of standards that are applicable to the code segment under inspection, whether or not testing was accepted as meeting the standard, or if the code needs testing and verification of standard acceptance.

2. All concepts to code mappings are referenced by a multitude of concepts, math indexing like "Mathematics Subject Classification" from zbMath so that people can find what they are looking for faster.

3. Any military, computer science CMMI ratings, ISO/IEC, IEEE, ASME, and other international standards relevance for the code segment under review.

All standards are clearly marked as verified and not verified, and whether additional testing is needed to make it applicable.

Here is what you are doing when you make code, you are doing all three at the same time:

1. A software code base made easier by concurrent version systems like CVS, git, and SourceForge;

2. A body of knowledge that is embodied in the code; and

3. A community of adherents (a group of people that follow a software knowledge base).

The Library in the Abstract (RULA)

This book is a extension of the ideas presented inside my novel library abstract, called RULA - the Re Use Library Abstraction. It's a prototype for the better way to leave code, not write code but plan, organize and configure to set all that it contains within a state of understanding that makes it very easy for others to improve upon the work. The goals are twofold: one to enlighten programmers into a better reality that they adopt because it's silly not to, but also to write an IEEE Standards Association Standard for the purpose of encapsulating all the rights and not-wrongs (in the double negation sense) into a more correct formalism (a word meant as positive here but taken as an invective later on to describe what is). The overall goal is to use this as a strawman to understand what works best and lead by example getting it standardized by an international standards body in good standing (for the audience, I was a member of the IEEE upon graduation from engineering college, but when the IEEE gave Bill Gates an honourary title for a $350 million donation, I refused to lower my standards).

Part I

Microscopic View: The Toolbox

Chapter 11

Pirate Treasure Map

You Are What You Have Been,
And What You Will Be Is What You Do Now
—Siddhartha Gautama, The Buddha

This section presents some ways to acquire code for your ultimate software reuse triumph. There are lots of sources, but knowing where to look quickly is hard unless you've failed at it many times. Learn and save your time from my failings. Like every other chapter in this tome, this is an opening and incomplete effort that will build towards perfection without attaining it over time. Check back later to the online software RULA for updates.

University computers

The profs that mentor, nurture, and educate the college undergrads also provide servers for their work, like a shingle on some office front, that deposit software as well as knowledge like conference papers, journals, and monographs like books as editors and career research areas, there's lots of booty to plunder. Yaarrgh.

SourceForge

"With the tools we provide, developers on SourceForge create powerful software in over 502,000 open source projects; we host over 2.1 million registered users. Our popular directory connects nearly 30 million visitors and serves more than 2.6 million software downloads a day."

"SourceForge is an Open Source community resource dedicated to helping open source projects be as successful as possible. We thrive on community collaboration to help us create a premier resource for open source software development and distribution."

"SourceForge is a complete business software and services comparison platform where buyers find, compare, review, and buy business software and IT services. Selling software? You're in the right place. We'll help you reach millions of intent-driven software and IT buyers and influencers every day, all day."

https://sourceforge.net

Github

Where the world builds software

Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.

https://github.com

Fedora SourceGraph

Recently. Fedora announced that SourceGraph (Figure 11.1) has made the entire repository of RPM (RedHat Package Manager) packages within the domain src.fedoraproject.org which, at the time of writing, had over 34,000 repositories.

They even describe the command that can help finding code:

The following query[1] *will scan all the repositories for software that is compatible with the "Open Source Definition" (OSD).*

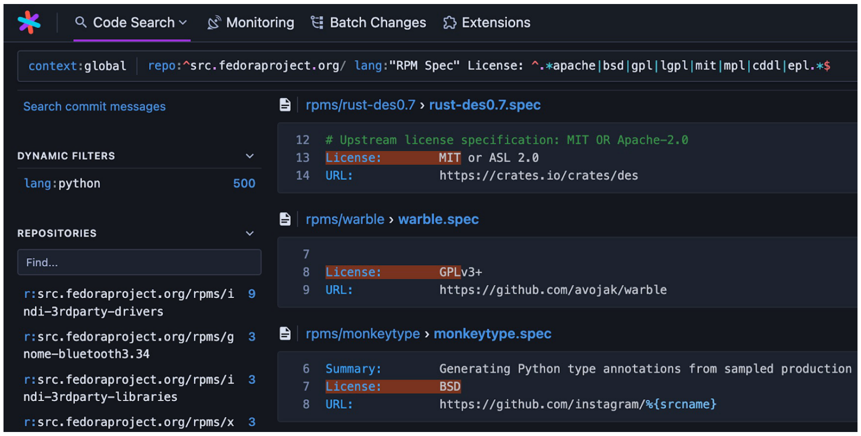
*repo: ^ src.fedoraproject.org/ lang:"RPM Spec" License: ^ .*apache|bsd|gpl|lgpl|mit|mpl|cddl|epl.*$*

https://fedoramagazine.org/using-sourcegraph-to-search-34000-fedora-repositories/

The Commonality

The Commonality is a quirky (with a very clear role), interesting JavaScript community called The Commonality on github. Of course, if you are anywhere near the sci-fi afficionado many nerds claim to be, then you will be aware that

[1] https://fedoramagazine.org/using-sourcegraph-to-search-34000-fedora-repositories/



```
context:global  repo:^src.fedoraproject.org/ lang:"RPM Spec" License: ^.*apache|bsd|gpl|lgpl|mit|mpl|cddl|epl.*$
```

r:src.fedoraproject.org/rpms/indi-3rdparty-drivers 9
r:src.fedoraproject.org/rpms/gnome-bluetooth3.34 3
r:src.fedoraproject.org/rpms/indi-3rdparty-libraries 3
r:src.fedoraproject.org/rpms/x 3

rpms/rust-des0.7 › rust-des0.7.spec

```
# Upstream license specification: MIT OR Apache-2.0
License:        MIT or ASL 2.0
URL:            https://crates.io/crates/des
```

rpms/warble › warble.spec

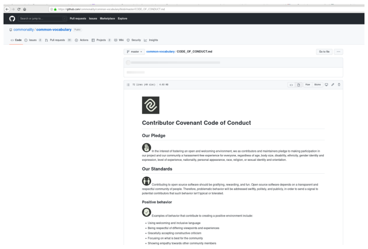
```

License:        GPLv3+
URL:            https://github.com/avojak/warble
```

rpms/monkeytype › monkeytype.spec

```
Summary:        Generating Python type annotations from sampled production
License:        BSD
URL:            https://github.com/instagram/%{srcname}
```

SourceGraph teams up with Fedora to search all Fedora repositories by licence.

“the commonality” was the Taelon’s hive mind link on Gene Roddenbery’s lesser known Earth: Final Conflict TV series. They have private members (mind out of the gutter, dude) and a code of conduct (Figure 11.2). They provide prototypes for others to use, but in order to commit you must meet the standards of the others. It seems like a very successful, thriving community of coders.


https://github.com/commonality/common-vocabulary/blob/master/CODE_OF_CONDUCT.md
Search or jump to...
Pull requests
Issues
Marketplace
Explore
commonality / common-vocabulary
Public
Code
Issues 2
Pull requests 31
Actions
Projects 2
Wiki
Security
Insights
master
common-vocabulary / CODE_OF_CONDUCT.md
Go to file
72 lines (48 sloc) 4.63 KB
Raw
Blame

Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Contributing to open source software should be gratifying, rewarding, and fun. Open source software depends on a transparent and respectful community of people. Therefore, problematic behavior will be addressed swiftly, politely, and publicly, in order to send a signal to potential contributors that such behavior isn't typical or tolerated.

Positive behavior

Examples of behavior that contribute to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

The Commonality Code of Conduct Page

advancedsourcecode.com

advancedsourcecode Luigi Rosa’s **advancedsourcecode.com** is an interesting one-inventor author site with a lot of advanced algorithms in both matlab and source code.

egroupware

egroupware is all you need if your software needs are very thin, like using a browser for some financial transactions, and the code is available with software as a service support options for those that would meet small teams without the need for heavy duty coders.

~

There are many more that can fill many volumes of a book but become obsolete and irrelevant if not upkept and pruned regularly. If I put in print a long list, I will hear from some that they were better and weren't included. This book presents but doesn't cover many topics. This section, in order to maintain currency and relevance, will be a vibrant augmented chapter over the future versions of this book. This section may be the reason to come back for updates.

Chapter 12

Reuse Tools

The Root Of Suffering
Is Attachments
—Siddhartha Gautama, The Buddha

Useful Tools you should learn about if you want to break into the game of software reuse are presented one per function and one example here in this chapter. I chose the free opensource ones but if you want someone to take your money for a tool that's only barely above the free one, a sucker is born any minute. That said, an enterprising person could take the advice of this book, improve the code of any one of these tools for a market and make a living on that.

Tools Identification

Looking outward sometimes needs to start by looking inward. For each activity in the existing or potential reuse process inside your organization, you can start at reuse by assessing your own tools. Get your team to describe how your existing tools might support reuse- perhaps because they process source code/documentation in a novel and perhaps proprietary way. Assess each potential tool's importance to reuse - do they provide your organization with a unique advantage? Assess the feasibility of existing tools - will retooling or reconfiguring these tools make a significant improvement in software reuse feasibility. Would automated tools drastically accelerate your reuse activity? Do you possess tools to assess the maturity development level of bespoke software?

Meld

meld

Meld is an open source tool that, as a GUI, allows you to compare and contrast files, including three-way comparison, and even more useful for reuse it can compare file directories and allows you to manipulate files and directories. It's a tool for those with no real preferred editor tools bent. It just works and it's free.

Here's how Meld the project and program describes itself:

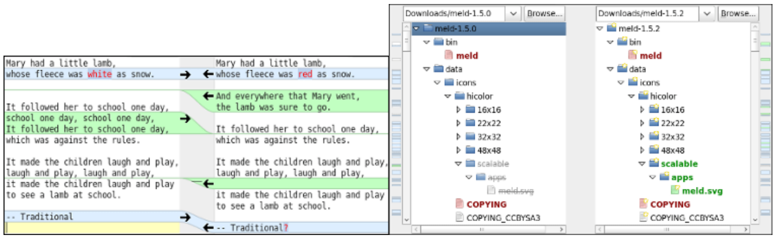
> Meld is a visual diff and merge tool targeted at developers. Meld helps you compare files, directories, and version controlled projects. It provides two- and three-way comparison of both files and directories, and has support for many popular version control systems.
>
> Meld helps you review code changes and understand patches. It might even help you to figure out what is going on in that merge you keep avoiding.

Meld has three modes of operation:

- File comparison
- Directory comparison
- Version control

> Edit files in-place, and your comparison updates on-the-fly Perform two- and three-way diffs and merges Easily navigate between differences and conflicts Visualize global and local differences with insertions, changes and conflicts marked Use the built-in regex text filtering to ignore uninteresting differences Syntax highlighting
>
> Directory comparison
>
> Compare two or three directories file-by-file, showing new, missing, and altered files Directly open file comparisons of any conflicting or differing files Filter out files or directories to avoid seeing spurious differences Simple file management is also available
>
> Version control
>
> Meld supports many version control systems, including Git, Mercurial, Bazaar and SVN Launch file comparisons to check what changes were made, before you commit View file versioning statuses Simple version control actions are also available (i.e., commit/update/add/remove/delete files)
>
> Get Started:

Here's how to meld differences between two files:



```
Mary had a little lamb,
whose fleece was white as snow.

It followed her to school one day,
school one day, school one day,
It followed her to school one day,
which was against the rules.

It made the children laugh and play,
laugh and play, laugh and play,
it made the children laugh and play
to see a lamb at school.

-- Traditional
```

```
Mary had a little lamb,
whose fleece was red as snow.

And everywhere that Mary went,
the lamb was sure to go.

It followed her to school one day,
which was against the rules.

It made the children laugh and play,
laugh and play, laugh and play,

it made the children laugh and play
to see a lamb at school.

-- Traditional?
```

```
Downloads/meld-1.5.0
meld-1.5.0
  bin
    meld
  data
    icons
      hicolor
        16x16
        22x22
        32x32
        48x48
        scalable
          apps
            ~~meld.svg~~
      COPYING
      COPYING_CCBYSA3
```

```
Downloads/meld-1.5.2
meld-1.5.2
  bin
    meld
  data
    icons
      hicolor
        16x16
        22x22
        32x32
        48x48
        scalable
          apps
            meld.svg
      COPYING
      COPYING_CCBYSA3
```

Meld: It works visually on both files and directories like diff and comp work on the command line shell.

Getting started comparing files

[1] lets you compare two or three text files side-by-side. You can start a new file comparison by selecting the File ▶ New... menu item.

Once you've selected your files, Meld will show them side-by-side. Differences between the files will be highlighted to make individual changes easier to see. Editing the files will cause the comparison to update on-the-fly. For details on navigating between individual changes, and on how to use change-based editing, see Dealing with changes. Meld's file comparisons

There are several different parts to a file comparison. The most important parts are the editors where your files appear. In addition to these editors, the areas around and between your files give you a visual overview and actions to help you handle changes between the files.

On the left and right-hand sides of the window, there are two small vertical bars showing various coloured blocks. These bars are designed to give you an overview of all of the differences between your two files. Each coloured block represents a section that is inserted, deleted, changed or in conflict between your files, depending on the block's colour used.

In between each pair of files is a segment that shows how the changed sections between your files correspond to each other. You can click on the arrows in a segment to replace sections in one file with sections from the other. You can also delete, copy or merge changes. For details on what you can do with individual change segments, see Changing changes. Saving your changes

Once you've finished editing your files, you need to save each file you've changed.

You can tell whether your files have been saved since they last changed by the save icon that appears next to the file name above each file. Also, the notebook label will show an asterisk (*) after any file that hasn't been saved.

[1] http://meldmerge.org/help/file-mode.html

You can save the current file by selecting the File ▶ Save menu item, or using the Ctrl+S keyboard shortcut.

Dealing with changes

Meld deals with differences between files as a list of change blocks or more simply changes. Each change is a group of lines which correspond between files. Since these changes are what you're usually interested in, Meld gives you specific tools to navigate between these changes and to edit them. You can find these tools in the Changes menu. Navigating between changes

You can navigate between changes with the Changes ▶ Previous change and Changes ▶ Next change menu items. You can also use your mouse's scroll wheel to move between changes, by scrolling on the central change bar.

Changing changes

In addition to directly editing text files, Meld gives you tools to move, copy or delete individual differences between files. The bar between two files not only shows you what parts of the two files correspond, but also lets you selectively merge or delete differing changes by clicking the arrow or cross icons next to the start of each change.

The default action is replace. This action replaces the contents of the corresponding change with the current change.

Hold down the Shift key to change the current action to delete. This action deletes the current change.

Hold down the Ctrl key to change the current action to insert. This action inserts the current change above or below (as selected) the corresponding change.

GNU comp, diff & patch

comp
diff
diff3
patch

These tools are all advanced enough to need a full manual each. GNU gives you 4 tools in one manual. Here is a synopsis for them taken from GNU diffutils manual.

1 What Comparison Means

There are several ways to think about the differences between two files. One way to think of the differences is as a series of lines that were deleted from, inserted in, or changed in one file to produce the other file. diff compares two files line by line, finds groups of lines that differ, and reports each group of differing lines. It can report the differing lines in several formats, which have different purposes. GNU diff can show whether files are different without detailing the differences. It also provides ways to suppress certain kinds of differences that are not important to you. Most commonly, such differences are changes in the amount of white space between words or lines. diff also provides ways to suppress differences in alphabetic case or in lines that match a regular expression that you provide. These options can accumulate; for example, you can ignore changes in both white space and alphabetic case. Another way to think of the differences between two files is as a sequence of pairs of bytes that can be either identical or different. cmp reports the differences between two files byte by byte, instead of line by line. As a result, it is often more useful than diff for comparing binary files. For text files, cmp is useful mainly when you want to know only whether two files are identical, or whether one file is a prefix of the other. To illustrate the effect that considering changes byte by byte can have compared with considering them line by line, think of what happens if a single newline character is added to the beginning of a file. If that file is then compared with an otherwise identical file that lacks the newline at the beginning, diff will report that a blank line has been added to the file, while cmp will report that almost every byte of the two files differs. diff3 normally compares three input files line by line, finds groups of lines that differ, and reports each group of differing lines. Its output is designed to make it easy to inspect two different sets of changes to the same file. These commands compare input files without necessarily reading them. For example, if diff is asked simply to report whether two files differ, and it discovers that the files have different sizes, it need not read them to do its job.

File Hunks

When comparing two files, diff finds sequences of lines common to both files, interspersed with groups of differing lines called hunks. Comparing two identical files yields one sequence of common lines and no hunks, because no lines differ. Comparing two entirely different files yields no common lines and one large hunk that contains all lines of both files. In general, there are many ways to match up lines between two given files. diff tries to minimize the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines. For example, suppose the file F contains the three lines ‘a’, ‘b’, ‘c’, and the file G contains the same three lines in reverse order ‘c’, ‘b’, ‘a’.

If diff finds the line ‘c’ as common, then the command ‘diff F G’ produces this output:

```
1,2d0
< a
< b
3a2,3
> b
> a
```

But if diff notices the common line ‘b’ instead, it produces this output:

```
1c1
< a
---
> c
3c3
< c
---
> a
```

It is also possible to find ‘a’ as the common line. diff does not always find an optimal matching between the files; it takes shortcuts to run faster. But its output is usually close to the shortest possible. You can adjust this tradeoff with the --minimal (-d) option (see Chapter 6 [diff Performance Trade-offs], page 33).

Suppressing Differences in Blank and Tab Spacing

The --ignore-tab-expansion (-E) option ignores the distinction between tabs and spaces on input. A tab is considered to be equivalent to the number of spaces to the next tab stop . The --ignore-trailing-space (-Z) option ignores white space at line end. The --ignore-space-change (-b) option is stronger than -E and -Z combined. It ignores white space at line end, and considers all other sequences of one or more white space characters within a line to be equivalent. With this option, diff considers the following two lines to be equivalent, where ‘$’ denotes the line end:

```
Here lyeth  muche rychnesse  in lytell space.   -- John Heywood$
Here lyeth muche rychnesse in lytell space. -- John Heywood   $
```

The --ignore-all-space (-w) option is stronger still. It ignores differences even if one line has white space where the other line has none. White space characters include tab, vertical tab, form feed, carriage return, and space; some locales may define additional characters to be white space. With this option, diff considers the following two lines to be equivalent, where ‘$’ denotes the line end and ‘^M’ denotes a carriage return:

```
Here lyeth  muche  rychnesse in lytell space.--  John Heywood$
  He relyeth much erychnes  seinly tells pace.  --John Heywood   ^M$
```

For many other programs newline is also a white space character, but diff is a line- oriented program and a newline character always ends a line. Hence the -w or --ignore- all-space option does not ignore newline-related changes; it ignores only other white space changes.

comp

Invoking cmp

The cmp command compares two files, and if they differ, tells the first byte and line number where they differ or reports that one file is a prefix of the other. Bytes and lines are numbered starting with 1. The arguments of cmp are as follows:

```
cmp options… from-file [to-file [from-skip [to-skip]]]
```

The file name - is always the standard input. cmp also uses the standard input if one file name is omitted. The from-skip and to-skip operands specify how many bytes to ignore at the start of each file; they are equivalent to the --ignore-initial=from-skip:to-skip option. By default, cmp outputs nothing if the two files have the same contents. If the two files have bytes that differ, cmp reports the location of the first difference to standard output:

```
from-file to-file differ: char byte-number, line line-number
```

If one file is a prefix of the other, cmp reports the shorter file's name to standard error, followed by a blank and extra information about the shorter file:

```
cmp: EOF on shorter-file extra-info
```

The message formats can differ outside the POSIX locale. POSIX allows but does not require the EOF diagnostic's file name to be followed by a blank and additional information. An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

diff

Invoking diff

The format for running the diff command is:

```
diff options… files…
```

In the simplest case, two file names from-file and to-file are given, and diff compares the contents of from-file and to-file. A file name of - stands for the standard input. If one file is a directory and the other is not, diff compares the file in the directory whose name is that of the non-directory. The non-directory file must not be -. If two file names are given and both are directories, diff compares corresponding files in both directories, in alphabetical order; this comparison is not recursive unless the --recursive (-r) option is given. diff never compares the actual contents of a directory as if it were a file. The file that is fully specified may not be standard input, because standard input is nameless and the notion of “file with the same name” does not apply. If the --from-file=file option is given, the number of file names is arbitrary, and file is compared to each named file. Similarly, if the --to-file=file option is given, each named file is compared to file. diff options begin with ‘-’, so normally file names may not begin with ‘-’. However, -- as an argument by itself treats the remaining arguments as file names even if they begin with ‘-’. An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

patch

Invoking patch

Normally patch is invoked like this:

```
patch <patchfile
```

The full format for invoking patch is:

```
patch options… [origfile [patchfile]]
```

You can also specify where to read the patch from with the -i patchfile or --input=patchfile option. If you do not specify patchfile, or if patchfile is -, patch reads the patch (that is, the diff output) from the standard input. If you do not specify an input file on the command line, patch tries to intuit from the leading text (any text in the patch that comes before the diff output) which file to edit. See Section 10.6 [Multiple Patches], page 49. By default, patch replaces the original input file with the patched version, possibly after renaming the original file into a backup file (see Section 10.9 [Backup Names], page 50, for a description of how patch names backup files). You can also specify where to put the output with the -o file or --output=file option; however, do not use this option if file is one of the input files.

The console analogs of the GUI meld - which probably calls diff and patch from inside the GUI - are GNU diff and GNU patch. If you prefer console / shell operation, or require to do automatic changes like from your own concurrent version server (CVS).

Gnu Autotools

autoreconf

automake
autoconf
aclocal
libtool
configure
make

You will hear stories of how hard GNU Autotools are to use, and that isn't altogether untrue. In fact, most complaints about GNU Autotools are true.

I will write this introduction with a very different slant and context than the other tool sections because I am an expert in these. That is because, apart from learning the syntax and the general mode of operation, learning to use GNU Autotools was the hardest conversion for me from my days as a Windows/VisualC++ programmer to a Unix/Autotools. There are two important facts that you need to take on board if you wish to be successful in reasonable timeframes: the files you are creating have various memory cached intermediate files that hold onto clagg and faulty configurations; if you don't purge them between runs, and often times restarting a new console with a fresh set of environment variables, will clear your problems. Here is another important advice: running **sudo updatedb** to update the files database tables after installing software will be needed for successful configuration; that will tackle most of the problems finding recent software you've just installed to satisfy missing requirements.

sudo updatedb

GNU Autotools allow you to configure all files you need to run compilation, or any other kind of recipe of making, with a custom and diverse kind of ways to leave most of the file checking, system information discovery, and compiler flags to the layer upon layer of scripts running and files made. I've used GNU Autotools to compile for embedded MCU boards, for multiprocessor CPUs, for a local network of agents, for the old Intercontinental Ballistic Missile (ICBM) Real-Time Operating System (RTOS) RTEMS (Real Time Executive for Missile Systems) and for computer architectures from 8-bit to 128 bit data buses. It can handle any variation you need for some custom application, and it hides nothing in any system it's applied to. I don't run doxygen, I use my pre-made **make dox** recipe so I don't have to remember the program flags to generate a generic Doxyfile (it's doxygen -g). You can make recipes to include a dynamic set of sources, you can combine scripting like BASH (Bourne Again SHell) scripts to execute programs and enter the returns back into your configurations.

make dox

doxygen -g

If I could describe the system in a simplistic overview, is it has a core script language in m4 that runs most of the configuration, it uses script-run files under all the programs that sponge up found configuration data, and that each program runs over the output of the last program to make it's own configurations. There are a myriad of software checks pre-made that if you spend the time looking for can solve you learning m4 and making your own scripts. Look in open source GNU Autotools project for a folder called m4 - it holds premade m4 scripts. Any open source project with an Makefile.am, a configure.in (old name) or a configure.ac, a Makefile.in or a autogen.sh, build.sh and so on file may have what you need.

Here's how GNU Autotools described in the itself :

> If you are new to Automake, maybe you know that it is part of a set of tools called The Autotools. Maybe you've already delved into a package full of files named configure, configure.ac, Makefile.in, Makefile.am, aclocal.m4, . . . , some of them claiming to be generated by Autoconf or Automake. But the exact purpose of these files and their relations is probably fuzzy. The goal of this chapter is to introduce you to this machinery, to show you how it works and how powerful it is. If you've never installed or seen such a package, do not worry: this chapter will walk you through it.
>
> If you need some teaching material, more illustrations, or a less automake-centered continuation, some slides for this introduction are available in Alexandre Duret-Lutz's Autotools Tutorial. This chapter is the written version of the first part of his tutorial.

https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

The book, GNU Autoconf, Automake, and Libtool by Gary Vaughn, Ben Elliston, Tom Tromey, and Ian Lance Taylor that does a good job, if you are thorough on setting up a proper autotools configuration system. I have used autotools on many projects, since 2003, and I assure you it will work just fine if you understand what they are made for and what When all else fails, and it will fail repeatedly, the testing regime will become the mainline to improve your awareness, your understanding of the components, and your ability to experiment with the changing how it works, and the testing regime will become the mainline of your advancements. You will save the most:

- Most code understandings;

- Most lost effort saved; and
- Most lost compile times.

Concurrent Version Systems

cvs

There are many concurrent version systems. There's CVS, subversion (svn), and of course gitlab with their new program gh. We will discuss just cvs here to make you aware of the kinds of command. CVS manuals can be found at GNU CVS . Here is their Introduction:

> *CVS is a version control system. Using it, you can record the history of your source files.*
>
> *For example, bugs sometimes creep in when software is modified, and you might not detect the bug until a long time after you make the modification. With CVS, you can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.*
>
> *You could of course save every version of every file you have ever created. This would however waste an enormous amount of disk space. CVS stores all the versions of a file in a single file in a clever way that only stores the differences between versions.*
>
> *CVS also helps you if you are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless you are extremely careful. Some editors, like GNU Emacs, try to make sure that two people never modify the same file at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.*
>
> *CVS started out as a bunch of shell scripts written by Dick Grune, posted to the newsgroup comp.sources.unix in the volume 6 release of July, 1986. While no actual code from these shell scripts is present in the current version of CVS much of the CVS conflict resolution algorithms come from them.*
>
> *In April, 1989, Brian Berliner designed and coded CVS. Jeff Polk later helped Brian with the design of the CVS module and vendor branch support.*

> *You can get CVS in a variety of ways, including free download from the Internet. For more information on downloading CVS and other CVS topics, see:*
>
> *http://cvs.nongnu.org/*
>
> *There is a mailing list, known as info-cvs@nongnu.org, devoted to CVS. To subscribe or unsubscribe write to info-cvs-request@nongnu.org. If you prefer a Usenet group, there is a one-way mirror (posts to the email list are usually sent to the news group, but not vice versa) of info-cvs@nongnu.org at news:gnu.cvs.help. The right Usenet group for posts is news:comp.software.config-mgmt which is for CVS discussions (along with other configuration management systems). In the future, it might be possible to create a comp.software.config-mgmt.cvs, but probably only if there is sufficient CVS traffic on news:comp.software.config-mgmt.*
>
> *You can also subscribe to the bug-cvs@nongnu.org mailing list, described in more detail in BUGS. To subscribe send mail to bug-cvs-request@nongnu.org. There is a two-way Usenet mirror (posts to the Usenet group are usually sent to the email list and vice versa) of bug-cvs@nongnu.org named news:gnu.cvs.bug.*

Astyle

astyle

I found Astyle[2] (Artistic Style 2.06) in the BRL-CAD source distribution and I immediately fell in love with it. It is a very well-written C++ code freshener and beautifier by Jim Pattee jimp03@email.com in 2016. You will note from elsewhere I have a real hard time with OOP like C++ but this one obeys the Unix Philosophy. It just understands a set of format requirements and plods through your code effortlessly and wisely then outputs both a newly formatted file in the identical name and an old copy suffixed with .orig so there's no pre nor post confusion. I added Jim's software to my RULA exemplar LoggerD project later on in the book.

Artistic Style is a source code indenter, formatter, and beautifier for the C, C++, C++/CLI, Objective-C, C# and Java programming languages.

When indenting source code, we as programmers have a tendency to use both spaces and tab characters to create the wanted indentation. Moreover, some editors by default insert spaces instead of tabs when pressing the tab key. Other editors (Emacs for example) have the ability to "pretty up" lines by

[2] http://astyle.sourceforge.net/

automatically setting up the white space before the code on the line, possibly inserting spaces in code that up to now used only tabs for indentation.

The NUMBER of spaces for each tab character in the source code can change between editors (unless the user sets up the number to his liking...). One of the standard problems programmers face when moving from one editor to another is that code containing both spaces and tabs, which was perfectly indented, suddenly becomes a mess to look at. Even if you as a programmer take care to ONLY use spaces or tabs, looking at other people's source code can still be problematic.

To address this problem, Artistic Style was created – a filter written in C++ that automatically re-indents and re-formats C / C++ / Objective-C / C++/CLI / C# / Java source files. It can be used from a command line, or it can be incorporated as a library in another program.

Doxygen

doxygen

Doxygen[3] makes automatic documentation of any software code base, effortlessly out of the box.

According to Doxygen:

> *Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL and to some extent D.*
>
> *Doxygen can help you in three ways:*
>
> *It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in $\mbox{\LaTeX}$) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically. You can also use*

[3] https://www.doxygen.nl/index.html

> *doxygen for creating normal documentation (as I did for the doxygen user manual and web-site).*
>
> *Doxygen is developed under Mac OS X and Linux, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available.*[4]

CGIF

ISO/IEC 247707:2007 Conceptual Graph Interchange Format (CGIF) is the way to embody layers of complexity without adding to the stratifications of the code itself. The following features are essential to the design of this ISO International Standard:

• Languages in the family have declarative semantics. It is possible to understand the meaning of expressions in these languages without appeal to an interpreter for manipulating those expressions.

• Languages in the family are logically comprehensive — at its most general, they provide for the expression of arbitrary first-order logical sentences. • Interchange of information among heterogeneous computer systems.

The following are within the scope of this International Standard:

• representation of information in ontologies and knowledge bases;

• specification of expressions that are the input or output of inference engines;

• formal interpretations of the symbols in the language.

The basic idea is to encapsulate the knowledge in your code **in a useable form that others can use to determine if your code meets their needs.** This is the backend to the

Check Your Knowledge!

Of course, as a way to gauge the quality of what you might find, you will look at the quality not of the code directly, but all the accompanying documentation, the organization, and the testing suite, more especially. You are better off determining how much knowledge lies within before you read a every line of code. You are going to scan for importaYou can find many things within that are valuable in their own right:

- Algorithms;

[4] https://www.doxygen.nl/index.html

- Models;
- Object Names;
- Threading or Forking;
- Math Error Codes; and
- Working examples:
  - Interrupts Polling/ Event driven code;
  - Message Marshaling and marshaling protocols compression and decompression techniques (hashing ...);
  - Custom hardware machine code, and machine language optimizations;
  - Debuggers Formats;
  - Visualization snippets like fragment shaders, vertex shaders; and
  - Reverse engineering embedded in code, and so on...

The skill set needed to leverage foreign code, unknown ideas inside the developer's heads, will rest within the honest assessment of the individual, the ideas within this tome are fit for the function of proscribing what you will find. This book counsels the wise to find the things that can be done and how to do them in a logical methodical way. It repeats concepts for the familiarized as things to heighten the awareness the vital things from a personal perspective.

What are the knowledge fundamentals:

- Understanding;
- Aesthetics;
- Optimizations (Size / Speed);
- Documentation Manuals, API, Readme;
- Organization; and
- Background Knowledge.

If people haven't made it clear, then how likely are you to find what you seek and what to keep in mind while searching? Herein are the major causes of your future misunderstandings.

In addition, key complaints you will discover when you read badly written code:

Definitions

Of all the things that will trip you up is the way in which what seems clear is not. The definition of words is the first of the many misunderstandings. Many teachers present the same topics in many ways. Many schools oversimplify notation and nomenclature. You will find this is at the heart of many assumptions about how things are laid out, and what they mean. Worse, if they were NEVER taught properly how to describe things, then you are in for a long drawn out deciphering. When terms are wrong, when variables names aren't what they should be, they make you make mistakes from an assumption you would otherwise avoid. This is one of the great struggles of epistemology, how to transform one unclear thing into a clear thing with the faulty means available, the lacking context, the missing flow, and so on. The error lies without (not included), the misunderstanding lies within.

Formalisms

The definition of formalism is the strict adherence to a given form of conduct, practice (remember your adherents are those that follow your conducts and practices). By this term, I mean the way the code is written. The quirky or offbeat way they define things, like how they declare classes and functions across headers and class files, the way they hide static variables, hidden defines that pop up to change what is written, their unique and nonstandard ways to declare configuration changes, the way they layout directories, the odd hidden files that impact everything but can't be seen. The illogical the build manner, and especially the integrated building into a GUI without any other options, the forced inclusion of rare, unheard of, and nonstandard libraries that don't give you any way to use the code configured without them.

Sloppiness - I must make this as the harshest and most obvious criticism of any code, sight unseen. Allow me to make the first and foremost apologies for the work of others. People write code rushed, they make shortcuts. They over promise, and under deliver. They have sick kids, and sick dogs, dental appointments and more meetings than they can suffer. The shortcuts lead into the folly of half-hearted and poor showings. You will find that the one thing you will bring into a disarrayed mess is the beginnings of order within the chaotic. If there is one recommendation that will help you in your hour of need, it is the realization and acceptance that you must order things as you progress. You will save yourself the trials by hacheting into the fragments and make a new start as you go. The faster you transmogrify what you find into what you need the sooner you will come to the state where it is satisfactory to you purpose.

Arcane Correctness

The most befuddling thing you will find are where in those unique occasions someone has a unique and arcane way of code. There are some people that learned the most correct yet unusual ways to code, special rules in the deep dark bowels of every programming language. They have what will look like the most incorrect way of handling things but it is just not common, yet perfectly correct. To find out, you will have to open your old manuals, your textbooks and find that it is in fact valid code. It may take some old programming magazine, some long in the tooth professor, and so on.

If you keep just the headers of this section handy, once you are finished this fantastical book, then you will be nearer to quantifying your difficulties into a strict category for deciding what kind of problem you've run into. Remember, they understood it when they coded it. You are left to understand what.

The Compiler is always right - Trust Your Instruments

If you went to a reputable college of any stripe, you would have had one or more humility speeches about always assuming the error is yours. If that's not the right kind of plea for you, then consider these wise words. As my grandfather, the WWII pilot decorated by King George VI with a Distinguished Flying Cross - for bringing back an airplane with one functional wing - used to say, "always trust your instruments." He lived by that for the entire WWII conflict, surviving three tours of duty over Europe. He said sometimes pilots would doubt their instruments at their peril: they would doubt their instruments, somewhere over ocean when separated from the group and they wouldn't come back.

When you are reusing code, it's never a wise attitude to doubt the automated programs that compile or interpret the source code and your fixes. These programs are made and operate successfully on any code base. If the errors are spat out about something wrong, it is the reused code that's the cause 99 times out of 100.

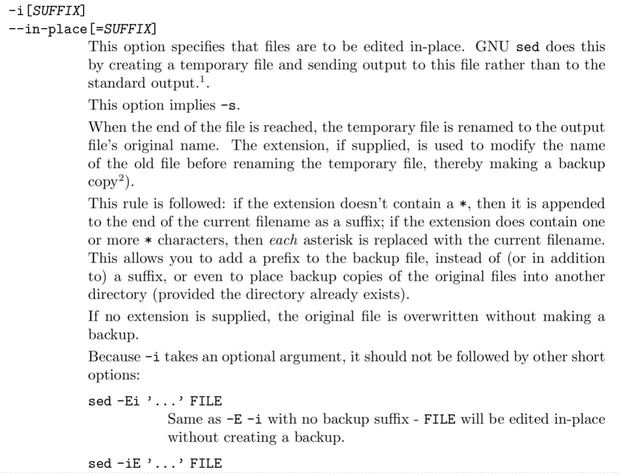
Documentation can speed you up...

The first way you limit the leveraging work is to read what documentation there is. The documentation, or lack there of, gives you insight in most cases to the quality of the code. There are some big exceptions, Ken Silverman's code is extremely dense in efficient code that was intended for his mind alone. I contacted him, to get some sense of his voxel code, and he was busy working with a company and gave me permission to do whatever I wanted with his code, but also he was far too busy working on his next great thing and unfortunately for me he was unavailable for comments. You will find many lone geniuses that are gratified to find people using the work they leave but are too fixated on their next big problem to bother you. Sadly, his code was hard to read and his test examples only worked on a Windows platform. I spent many hours just converting it to Unix worthy linking code. I also set it aside because it didn't fit my graphical needs but I will get back to it at some point.

Documentation is important, and rushed coders often neglect the non-obviousisms that they already know. Case in point: sed. GNU's sed (stream editor) is a nice little unix program to process files so you never have to write your own code if you can run sed for free. However, the documentation is opaque to people that aren't "in the know". Example,

sed

> *-i[SUFFIX] --in-place[=SUFFIX]*
>
> *This option specifies that files are to be edited in-place. GNU sed does this by creating a temporary file*
>
> *and sending output to this file rather than to the standard output.*
>
> *This option implies -s.*
>
> *When the end of the file is reached, the temporary file is renamed to the output files original name.*
>
> *The extension, if supplied, is used to modify the name of the old file before renaming the temporary file,*
>
> *thereby making a backup copy This rule is followed: if the extension doesn't contain a *, then it is*
>
> *appended to the end of the current filename as a suffix; if the extension does contain one or more * characters,*
>
> *then each asterisk is replaced with the current filename. This allows you to add a prefix to the backup file,*
>
> *instead of (or in addition to) a suffix, or even to place backup copies of the original files into another*
>
> *directory (provided the directory already exists).*
>
> *If no extension is supplied, the original file is overwritten without making a backup.*
>
> *Because -i takes an optional argument, it should not be followed by other short options:*
>
> *sed -Ei '...' FILE*
>
> *Same as -E -i with no backup suffix - FILE will be edited in-place without creating a backup.*
>
> *sed -iE '...' FILE*



`-i[`*`SUFFIX`*`]`
`--in-place[=`*`SUFFIX`*`]`

This option specifies that files are to be edited in-place. GNU `sed` does this by creating a temporary file and sending output to this file rather than to the standard output.[1].

This option implies `-s`.

When the end of the file is reached, the temporary file is renamed to the output file's original name. The extension, if supplied, is used to modify the name of the old file before renaming the temporary file, thereby making a backup copy[2]).

This rule is followed: if the extension doesn't contain a `*`, then it is appended to the end of the current filename as a suffix; if the extension does contain one or more `*` characters, then *each* asterisk is replaced with the current filename. This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix, or even to place backup copies of the original files into another directory (provided the directory already exists).

If no extension is supplied, the original file is overwritten without making a backup.

Because `-i` takes an optional argument, it should not be followed by other short options:

`sed -Ei '...' FILE`

Same as `-E -i` with no backup suffix - `FILE` will be edited in-place without creating a backup.

`sed -iE '...' FILE`

GNU sed manual, version 4.8, 1 January 2020, page 3

This is equivalent to --in-place=E, creating FILEE as backup of FILE

This is the verbatim wording of the explanation of the -i flag given on the command line to the sed program that has been around so long that it is now in version 4.8. It is precisely the reason why you can't find a straight answer to a simple question like "how do I save the file to a different output file with a different suffix?" like happens over 1000 times a day, 365 days a year. StackExchange is littered with thousands of questions that are essentially the SAME THING OVER AND OVER because the manual isn't clear. Do you notice the subtle change in the wording, one that is bound to cause confusion to people that just want to run a command once and get on with life? I was working on the software library RULA for the launch of this book and I wanted to set up some bash shellscripts to filter words like my bluntifying experiment but automatically. And I was forced to download the manual for myself because the ready answers show that most people don't know what this paragraph says (Figure 12.2).

Here's what it actually says: if you use the -i command (flag) and include a literal string (including punctuation!) right against the -i flag then when sed is done substituting for all the commands (because as it says that -i command IMPLIES -s command - you must do both) you gave it to filter your file for, it will automagically take that filtered buffer once complete and SAVE THE ORIGINAL

FILE NAME TO ANOTHER FILE WITH EXACTLY 'SUFFIX' CHARACTERS APPENDED TO THE END OF IT MAKING 'input.fileSUFFIX' that new file's name. THAT INCLUDES ONLY THE PUNCTUATION YOU SUPPLY, so -i.strip means the new filename (if the file was called filename.in) will become filename.in.strip'. The ability to change the name of a file is BAKED INTO sed. But by declaring it a SUFFIX and then talking for 3 paragraphs about an EXTENSION they confuse the point of what they are trying to say. No reasonable person can be blamed for not understanding exactly how to use this -i flag because the documentation authors change definitions of words (we declare EXTENSION is the same as SUFFIX) mid paragraph and they don't supply verbose examples to make the function clear.

When you are writing documentation, if you only say it once in one way, it's clear enough to you, but you've lost many people that don't see the implicit things you are assuming to be true (while reading this then assume EXTENSION and SUFFIX are the same concept!). More documentation is always better. More examples, more graphs, more cross-referencing, more definitions.

The only reason I pieced this together is I READ CHAPTER 6, which describes how sed functions BEFORE I read chapter 1. And then I ran a couple of sed examples in a BASH script and compared the output to see the revelation. I realized that in the process of making the final output, that sed makes a buffer that is essentially another file ready to go and save on it's own, it just needs to know where to put it. That would have been nice to explain on page 1, paragraph one.

So, if your reader is a sociology professor with no formal programming training nor education, wanting to collect and gather various words from his papers into a common file, and he isn't a programmer in the know about suffix and extension in computer-speak, it's not that he's stupid that he can't figure it out, it's you the documenter wasn't smart enough to make it clear, make it consistent, make it verbose.

Testing, Testing, Testing!

Like Pete Goodliffe draws attention to in his wise book, Becoming a Better Programmer[7], Chapter 25, that code testing is a lot like driver's tests for vehicles:

> *Driving is an interesting analogue of programming. Learning to drive has many parallels with learning our craft (software programmer), and there are lessons we can learn from a comparison of the two...to enter*

> *gainful employment you do have to demonstrate a reasonable level of skill: having passed a reputable training course, or showing tangible prior experience...Indeed, advancing skills can be orthogonal to the typical developer promotion path...But that doesn't mean necessarily mean that you're any better a programmer than when you started.*

So if documentation is sparse then if you want to invest significant time you need to backfill the understanding effort by testing regimes. I know people hate the GNU autoconf, automake, and make compilation tools but with make check as a recipe (that's what Richard Stallman calls his makefiles) included in many open source code libraries you will find it easy to make an external check system that can pass and fail areas of the code at any granularity you like.

The Use of a Debugger

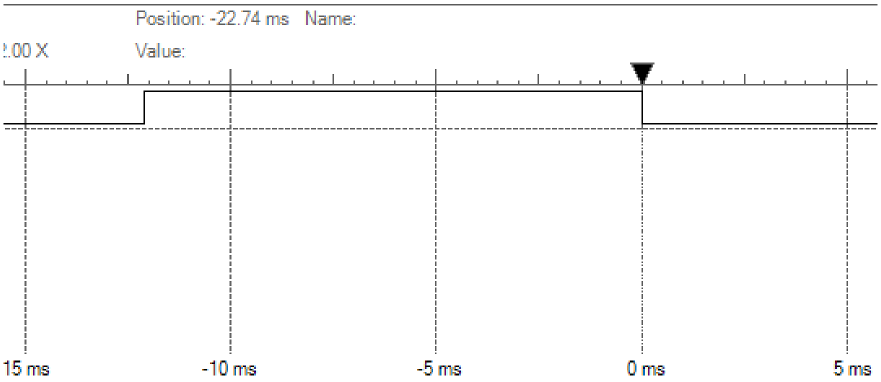
gdb

For many, a debugger is the way they learn about code. Debugging with GDB[13] by Richard Stallman, Roland Pesch, Stan Shebs *et. al.* is a definitive reference for the GNU Debugger.

I have, personally, found that in most cases you can learn as much, when you isolate to a single running process and dumping output in file form or to the command line with printfs. The reason I use this is because I like the lived experience of reading through the data to understand what is contained at every level. I find that make it easier to fix things at the bottom of the architecture. whether one uses fprintf to a file or just to the command line if the original programmers didn't have a DEBUG configuration defined it tells you a lot about how thoroughly they tested as they went. But to each his own.

Of course there's a performance hit for using printf inside code, it's blocking and per process as configured, that makes your process slow and it takes a hit when multiple threads access the printf function with blocking. Here's Jacob's from Beningo.com embedded printf (example in Figure 12.3) time response in an embedded microcontroller:

> *Using printf comes with a few problems that are often overlooked by developers. The first, a developer must bring in a standard C library that will undoubtedly increase ROM and RAM usage. Second, every time a printf stamen is used, the system becomes blocked until all characters have been transmitted which can result in significant real-time performance degradation. Take for example, a simple string to output such as "Hello World!" being printed out a*



Printing “Hello World!” From: https://www.beningo.com. © 2016 Jacob Beningo, All Rights Reserved. Used with permission.

> *UART at 9600 (still a very common occurrence). I performed a simple timing measurement on an STM32 and as shown in Figure 1, it took 12.5 milliseconds for the string to be formatted and printed to the terminal.Adding any string formatting makes the situation even worse! Printing the system state to the terminal using printf(“The system state is %d”, State) results in a 21 millisecond application delay as the string is formatted and transmitted. One might argue that running at 9600 baud is ridiculous but even increasing to 115200 would still result in 1.05 and 1.75 milliseconds respectively to transmit these two messages. A lot of processor bandwidth and potential real-time performance hits for minimally useful information.*[5]

A better way, for larger CPU based systems that can multi-thread, and medium sized MCU microcontroller units with limited multi-processing or multi-threading, is to use a logger thread.

The logger thread, rather than compartmentalize all for all processes, run another thread/process that has the collective job of writing out to file and/or the standard out (screen for a console). Then each process initiates a shared memory insert of a message to the stream, and continues with processing. This logger thread works in the background and gives you a single point of reporting. The messages consist of three parts - a warning, alert, or error code, a numerical code describing the kind of error, and string description of the error. Along with

[5] used with permission from https://www.beningo.com/getting-the-most-performance-from-printf/, Copyright 2016 Jacob Beningo

a logger timestamp, this forms a simple log file that can be read in real time with another process and tail function, or after the process ends.

The design of the test regime confirms how the software works when nothing else will do. There's a decision on how much to evaluate and the levels that you want to conduct testing (bottom level classes to top level engine classes). Ideally, you will dissect the code at the same level you want to test. But that's not always the case. I've used found that while that's idyllic, when you insert into the macro level and change how they work you will unwittingly make garbage data at a lower level. The goal of testing is understanding, nothing explains the data more than giving specific data into a code snippet and reading the outputs.

The first kind of tests you want are the ones that make it clear what data is set, especially the math settings and return gets. You need to figure out the simple things: is the data normalized? Are the angles in radians or degrees, or quaternions? Vectors are normalized or absolute? Are the positions in Cartesian coordinates, voxel distances, cylindrical or spherical coordinate reference frame data? Which way is up? What directions do the signs represent? Are positives absolute or relative? The second set of tests you will do cover all the ways you need to understand how pointers are sent. When you get different formalisms on how they describe code, some people have learned to make obfuscated types out of pointers and that will cause you to incorrectly assign pointers. You will cause lots of segmentation fault crashes if you are going quickly to convert code. The secondary problem with pointer misassignments is the improper size of dynamic memory allocations for the written objects. Crashes means misalignment of memory sizes from expected versus actual. You can use the equivalent of sizeof() functions available to see the proper sizes for member that should be exchanged to make the correct code. Void pointers are a way to get around one of the You need to understand what parts you need to construct and which parts are automatically created. Then, what are the ordering of what things need to be created and initialized before you can use them.

Once all the pieces are finished, you can then measure the performance metrics for use, including storage footprint size in memory and speed of operation. For these I recommend just using the compiler's own optimizations are first they tend to do a reasonable first effort at unrolling loops for faster operations and in the opposite direction reducing memory code occupies for the processor target you intend.

If you have tried a top down software leveraging strategy then you may arrive at success without a lot of time. But then the real problems will arrive later if and when you decide to take pieces out and reduce many similar data structs into a more efficient less redundant version. This kind of work is smiled upon in the Unix philosophy of coding, where it demands that you keep a single point of failure for all the important elements. If you have kludged together your code and the leveraged code without rewriting it, you will have some levels of data conversion from one side to the other; this kind of thing is the hallmark of where careless errors plague good code. While I urge you to keep messages the same, the internals will work better with a more efficient internal system.

Remember, the more data types you hold, the more testing you need to keep, and the more chances careless errors creep in. Single point of truth and single point of failure are valuable lessons.

The real problems happen when you try the same using in another mode, with a set of conditions that are secondary and not well documented, and traverse the parts of code you wallpapered over to get the parts you wanted working. There is no substitute for thorough understanding of the pieces exploded out in great detail. When you can't figure out where the data profession goes - where the chain of intermediate elements begin and end - you will be forced to break it all apart from the bottom up and get the data.

Full testing is exhaustive, and it often can't be full even at the time of production. I will go back to the story of protobuf-c as the example of noncomplete code testing. For many years, with intervals of a month or two, code that was presumed complete and working spits out error conditions, mainly from various alternative uses of the data types in ways that no one has used before. There's a flurry of patch and testing messages and emails between the developers and then the code is uprevved. This has gone on for as long as I've used the code in 2014. It's the reality that every code base goes through, even after beta testing. In a relative scale the code base isn't all that big, perhaps 20,000 LOC and still until those unusual conditions are met the errors remain beneath the surface. Novel bugs aren't novel, they exist in the code base from the end of code writing, and they surface when you cross their use in a novel way.

Chapter 13

General Methods of Improving Code

Let None Find Fault With Others;
Let None See The Omissions and Commissions of Others.
But Let One See One's Own Acts, Done and Undone.
—Siddhartha Gautama, The Buddha

When you go through a new source code download, it's kind of like you are looking into someone's dilapidated shed trying to find the hedge shears. You duck under a poorly hung light, push away the lamp cord, and then get down on your hands and knees, on all four, to gaze under a dark table and thence into an even darker corner. You know you will get lost, you know it will become frustrating, and you will start making changes, like shifting around stuff you will regret when you try to get back up again. It's exasperating, confusing, and time consuming (as I will mention a few times, you are saving 40%). It's very easy to complain about all the problems. But we don't know all the problems from the outset, the lack of knowledge, the needed bug fixes that lead to suspicion for the whole and doubt. Like the words I'm writing now. This book as it appears before me isn't ready for the spotlight. It's tragedy, the smaller the software team, the more a few people were forced to pick up the slack for others. Without any time for research, to look for part solutions that were working, you will find the problems they never got to. This book was a disordered mess until I read through it many times, edited, and critiqued as I went. Every time I did, I made a point to question my underlying assumption that this book does hold valuable insight, experience, and ideas that WILL help you improve your code harvesting and improved software performance.

Here's another insight, young people with the least amount of design experience, not perhaps design knowledge, are the ones that make the bad choices you wouldn't make with some seasoning and experience. Young developers leap above the cliffs, older people saunter up the goat path. On the other end of the spectrum, there are the experts. Those that have made their bones and can make good designs. But older ones like me are lazy, we get distracted being dads and moms, have real life problems like kids on drugs that intersect with the needs of the work. So if you can guess as to the average age of the developers then perhaps you can make a guesstimate of what you will find inside. Somewhere in the middle, are poseurs like university profs. They can talk your ear off about how well their project will work, but come the work they are off onto interesting stuff while they leave undergrads, teaching assistants and the odd post grad student to do the work. The big promises are lost once the funds for students are delivered, and results that are barely working becomes the standard of work. To be fair, if you are working for free as a student and you can finish early to get a job, there's no real motivation to provide the rest for free.

Either way, one must accept that the code you get for free is just as worthy of caution as the old Latin saying, CAVEAT EMPTOR. Let the buyer beware. So again, I estimate that when you take on board code you are saving yourself, at most, 40% effort. Of course, your results may vary.

Remember, sometimes people design code, then write code, and then they fix it. The documentation may be written at any time during that process, and most times never gets updated at the end. There will be obvious and nonobvious disconnects between them. There is no guarantee the process itself is serial. Often, the final working version may diverge dramatically from the documentation. What's the one way to bridge this misunderstanding gap? Testing, as we described earlier, the vital way you make your software show by it's performance and by it's own examples how and why to use it.

What are you doing when you renovate working software made by others? Essentially you find a bunch of wet pasta noodles hanging from a collander above a kitchen sink. These noodles are bundled together in a mess, with many of them sticking through the colander holes in such a way that you can't intuitively see where one noodle begins and under the bundle where it ends. As you spin around and point your finger, guiding it along many pasta noodle surfaces, you get confused and make wrong assumptions about which pasta noodle is which. People will tell you there's more to computer source code than memory and functions, but in reality it's just a differently labelled - a set of kinds and types which hold no real special purpose - memory addresses and functions. Many profs will fill your head with the niceties of software designs, that principles and techniques make your code better. At the end of the day they all boil down to a residue of memory as data and memory as instructions. Names, offsets, pointers and the like. At the abstract level, one is left with data; data of kinds without any meaning other than the locations allow (i.e. the data registers of an ethernet interface shows the address of the sender, the size of a data packet, and so on). So any reworking of the data types in software code is what your purpose is. If you change a function into a defined macro, you are saving the mangled names of the function inside the object directories of the compiled code, and copying the same macro over an over as data and data-instructions (machine code operators). This is the reality of code. Despite all the high minded meanings and ideals that some profs will fill your head with, they are merely figments of imagination and boil down to an underwhelming reside. The bells as whistles they proclaim are important aren't really all the special and significant, just read the innards of a C++ compiler to understand how little is changed from C. Yes it handles garbage collections and type safe operations, but you can do that all with ANSI C. Even the Grady and Booch design patterns boil down to lacking functionality inside C++ compilers that exist in other programming languages. The problem isn't just that few profs can't be honest about how little important extras exist within real code, most of these so-called teachers have never READ a compiler to understand it.

On the other hand, accepting the spartan reality of what you will find at the bottom of all the obfuscation and indirections, is that you can make a methodical conversion of code strategy, a leveraging code method as laid out below, to ease the transition for yourself and your team. Cut through all the classes, methods, machine learning (which merely learns to adapt some data based on other data in a time varying way), declarations and types and within the code you will see some valuable core things that can be kept and an abstract way to handle that that ignores the temporary meanings. What this book does is accept that it's all just kinds of memory inside the source code, code that in most cases gets compiled or interpreted code for interpreters, that can be repurposed. The kind of repurposings described herein are described by what kind of wholesale changes you make to the code itself. Remember, this book provides proscriptive guidance, and not descriptive help by way of a few examples because descriptive advice allows you to fill in the applicable details when and where you see that problem. If I provided descriptive advice (build a library no more than 500MB size), and those ways don't help what you will achieve, I run the risk of you doubting there's a better way and waste your time somehow. By leaving the implementation up to your mind, I will never be completely wrong even if I'm not 100% helpful. That's the right side of correct and vague as I would rather be than spectacularly insightful yet wrong. You will find many sentences end with the ambiguous “and so on”. This is on purpose. Instead of knowing at the outset what other members may be part of YOUR set members, I add vagaries to ideas so when your own ideas trigger a eureka moment, it’s your brilliance, fair and square. Hopefully, you will remember your brilliance the next time I release a book...

Here are logical, methodical ways to handle working code where you keep the overall strategy as a starting point, or perhaps a guide, so as you look into the various files you will have something to hold onto. Often I start with a doxygen to document the code, find little of value, and then start working away at some key files.

Design Patterns

The best way to go forward in computer science is to start from the well-designed computing patterns that exist as starting points for your work. For example, many people have made "lists" and the RULA library curates a couple. Design patterns like Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides "Design Patterns" have been scoffed at and mocked for being derivative but there’s no need making a new generator factory. Another source is Grady Booch’s Object-Oriented Analysis and Design with Applications. There are many pre-made forms to adopt. That saves lots of work.

Adopting pre-made design patterns to your own code is by itself is a considerable adoption of reuse

There are tons of pre-made, pre-thought patterns: cantrips, generators, bubble sorts, and on and on. Seek them out first.

The first technique you need to master after understanding a design pattern, is making it fit into your purpose. You need to be able to see the code as a starting point, and what needs to be added to make your inputs and outputs. My advice if you are starting out is just begin with other people’s patterns, make some mods and watch it work. Then as you gain experience you will imagine your known patterns overlaid on top of other’s code. Know what parts are lacking and need to be inserted.

One of the most helpful things is to just run code, with lots of debug info on, and you will learn where these patterns really work and where they don’t. When you get done, and run it in the field, you will see where an octree makes more sense than a hashed database entry, and so on.

All or Nothing Software Leveraging Methods

The following methods work better if you apply them to all the code you see, in that they don't require you to know dependencies to include and/or avoid when leveraging code. The follow-on hybrid code leverage methods will require more reading and understanding to pull off victoriously.

Bottom up

One struct and function at a time. One method, one class, and one file. One by one, take out and integrated into your existing code base.

Top down

You take over the code from the top, from the working example, like the GUI. You attempt to just look at the highest level classes / data structs and make an assessment of what you might get it to do other than the direct way it's intended to work in the provided application. If you want nothing more than to use the inputs and returned outputs with no specialization or customization then this is the way it works faster than bottom down

Hybrid Software Leveraging Methods

These kinds of methods attack the unknown and unseen knowledge within the code without a deliberate full-scale transformation in mind. These can be methodical translations like documentary, or reinterpretations like Reidentify.

The aim of this section is to put before your mind the ideas that are separate and distinct from the source code itself, but by placing these ideas in closer proximity to the reuse ideas, you can see your way clear to making it all better. Remember, this isn't a software problem without being an epistemological problem also. Knowledge and software are inextricable, and you can maximize one by applying both.

Extraction

Full Partial Single Harvest Algorithm extraction - If you wish nothing to do with the code, you look for the main mathematical functions, you simply printout out the core of that software code, and you read the algorithm from the lines. It's far harder to read it if the programmers used a different formalism than you are used to. If the code depends on other libraries like standard math libraries you will probably need to run a test program to understand the code.

ReProcess -

Change alter IO Same Functions different data Same Data, Different functions

Extract the algorithms

Lobotomizing -

The most extreme form of software reuse you can do is what I call lobotomizing. Imagine a fully connected and properly designed software library works like a head and body controlled by nerves. In a lobotomy you remove as much of the brain and nervous system as it takes for the patient. In this case, you rip out the higher data and function types that form the brain and nerves in data and functions. You crack the skull, latch onto the brain with forceps and pull it all out of the body - nerves and all. You are left with the hollowed corpse of a software library. This is drastic as the surgery goes, but it is also a safe way to reduce the area of bugs and errors. If you have an existing proven code base like a database system then you both isolate which side the errors are mostly on, and what adjustments you will need to fit the new code into the old nerve locations. This is tedious and it's painstaking. It's easy to get lost reattaching function over function. Sometimes it's better to divide and conquer by just taking a small extraction one at a time. I know the paradigm presented isn't a precise one, it's barely adequate, but it evokes some powerful imagery that will stay with you when making your redesign decisions.

Documentary

Document everything. Explain it all. Make the data flows clear for people to read and use. Make special pre-fix and post-fix characters for variable typedefs, and so on. Use WordsSmashedTogether for longer and clearer function names. Use time-worn function/method names and concepts like Get/Set (returning a variable and setting the internal variable. This is the way to make what is unclear clear.

Reidentify

The point of an identity is that it makes some sense to you. You don't know about nouns you've never experienced, nor read, nor heard of. There must be a familiarity to the things you create within the limits of your knowledge and experience.

If you think better in physics terms than computer programming terminologs (terms of computer programming, not the study of terminology), then adapt the names of things from Richard Feynman's quaint Quantum Electro Dynamics ideas[5] instead (instead of deconstructors, call them annihilators!).

Take things that you don’t get, and rename them to something within your knowledge wheelhouse. If you don’t like the function and anti-functions (an anti-function reverses the function process - an integral is also known in some cases as the anti-derivative) then rename them and make the necessary functor changes so they make and change the state of things inside your code more to the liking inside your head. Remember, no matter how good your code is, it can always be more clear and more logical, more linear, and more presentable.

Like design patterns, you would be better off adopting the common ones for common code functions / functors like Get/Set because that will make it easier for others to adapt to your work because it makes similar concepts similar with intuition and no extra processing needed. Standards are standard for a reason. Of course, if you are the first to make a new standard then it behooves your community to spend extra effort making it in a welcoming form for others to get excited about using.

If names and concepts aren’t cutting it, you should try other methods to clear the code, draw figures, diagrams, make ontologies of things and so on to get the message across in many ways.

Standardizing

Standardizing is the general actions that remove flawed, quaint, unique, and parts in favour of more specific standards that can be tested to an international standard, like better tested replacements like the GSL - Gnu Scientific Library for a better version. One of the best examples is math, many people cook their own matrix math algorithms, and they can be replaced with better, faster math like in the case of LAPACK/ ATLAS which are customized, optimized linear algebra FORTRAN routines that compute faster, there for better able to make other code faster. Special code snippets for workarounds in libraries that existed is also another kind of this effort. Another example is some old propriety code from insides a Sun Microsystems Unix version or a library from Silicon Graphics. The most common would be extracting old, poorly documented for a concise, and safer, versions. Most of this work will be more tidying and improving versus dramatically changing the way the code works.

There are standards going out in every direction for software, from requirements like the POSIX.4 [6] for time-critical code, compiling from a standard compiler like GCC [8, 14]. It could be implementing code to work within embedded systems like ISO/IEC 18037:2004[9]. It could be standardizing your GUI on the Qt multi-platform C++ code.

It’s a common find to see ways to improve things that gets put on the list of things you plan to get to before you retire.

Commonifying

Another way to fuse code from many sources to work inside their own solitudes. In this method, one takes the interfaces as they were written in, downgrade them to the basic data types, or to convert from one to another via a common messaging system like TAO (The ACE Orb), a middleware, and so on. Run each as separate process and make them read and write from each other, then you need to prepare for race conditions, arbitration, and bottlenecks in messages. This is like removing duplications that will appear, but may or may not replace everything.

You might have a cmake, a GNU autotools, and BASH portable shellscript[12] install systems into portable shell scripting alone. And so on.

Sam Newman uses the acronym DRY: don't repeat yourself. He meant don't repeat your system behaviour and state information, but it's equally useful when you think of three separate math functions to handle the same equations from different source codes. Many old codes have their own matrix multiplications. I haul them out and apply ATLAS because it can be compiled faster on any machine. This is also recommended by the The Art of Unix Programming: SPOF - Single Point of Failure and SPOT - Single Point of Truth. You make one part hold one truth and you make another point transform the truth anywhere else. Have no doubt where the errors lie.

Here's another way to make code from other

Scion

A scion is like the plant grafting technique. You take out a chunk of code and place it within your existing code base. You cut it out of the API and replace it within. A lot of things can be moved over, but what involved knowing exactly where that part begins, processes, and ends.

A scion requires you to understand the internals in as much as you keep the code together and send it and receive from it the members and methods it expects. You don't change it at all. If it relies on external libraries, you include them. You don't adapt the code, you don't change your code, you just make your interface handle the internals of the scion. The scion may contain levels you haven't read, don't understand or don't have time to. It's like all those sci-fi movies where they magically take the warp engines from one ship and insert them where a broken one existed.

The hallmark of the scion is that you will have to chop through many files, headers, source code etc., and that the beginning and end of the code will involve cutting through many files, perhaps an entire library, to isolate them.

Minimal Interface Mechanism (MIM)

By this method you hide code within code program within a process or thread and you send to it and receive from it low level data types like floats, ints, char strings, and doubles. You touch nothing within the code, you interact and interface indirectly through a couple of translation functions like TranslateIn - to translate into the data types internal to the code and Translate out - to convert data to simple types and return them by some method - shared data, mutexes, etc. - to and from the independent code. The kind of interfacing is very simple and only needs to read a few class file declarations to determine what will be passed through. Now, this may seem like a perfectly valid way to operate, but it doesn't guarantee that the data formatted into common types is the right endedness, the correct width for the CPU address architecture. It also doesn't guarantee you will find the east spots to cleft the code and get exactly what you want. As you peel back layers, you may delay and invariably run into massive changes to make a "simple" interface work. It might mean serious reorganization, and so on.

The mechanism for sending and receiving minimal interface data is up to you, daisy chain processes, event driven interrupts, Chinese fire drill (buckets passed onto the next process starting from input to the current process to output returned elsewhere).

Of course, this kind of leveraging strategy makes no guarantees it will be fast enough or resilient enough, and may result in manual overhauls of other software to make them work together. The more code you don't convert leaves an unknown risk down the road to deal with.

Chapter 14

Logger Reuse Code Exemplar

Peace Comes From Within.
Do Not Seek It Without.
—Siddhartha Gautama, The Buddha

Enough talk, here's some action. This is a very trivial example of code reuse. It sits apart from the rest of the code inside my library code for the purpose of using this as a basic example of how to reuse other's code. It has three folders inside "originals" folder to copy and keep all the files you can go back and pull more code out This is basically Michael Kerrisk's two exemplar source folders melded into one more complex working daemon. This reuse example is easy for two reasons: one, both code bases originate from the same programmer so the thought model is identical and it's well documented. It's obvious that Mike teaches Linux because he focuses on the inner workings by varying topic set out as isolated examples.

One warning I need to explain at the forefront of this knowledge is that this is very intensive with the GNU Autotools configuration system. While this book isn't about GNU autotools you can in your mind separate from the strategic reuse decisions I make in this chapter a delineated reuse strategy if you just ignore the configuration stuff. There is a line of reasoning and I make it plain. You don't need to be a GNU Autotools expert to see the path, but if you want to use this toolbox I am an expert at applying these tools, after nearly 20 years using them. So how and why I use commands is legit, even if you don't get them all to work at the first attempt. GNU Autotools are frustrating, take a break, retrace some steps and you will eventually get there. I will include some extra comments about GNU Autotools in the file FAQ. Now, back to the main thread...

This logger will work on Unix and Linux systems with the right kernel range. I used Fedora 28 on both machines I compiled it on. I am using an older kernel as I tend to do work on my machines, I don't spend time washing the tires and tuning it up.

What this logger does is run in the background on your system as a daemon. Once it is running, you can send syslog messages onto the logger. You need to decide when to run it and I haven't set it up to run from startup but as a user run daemon that you decide to run it. This can be changed if you know how to run programs within the system startup (like etc/crond cron daemon) daemons. It will run without a console / tty terminal and it's other input and output are set to the null device.

The idea here is you are using the system the way it was meant to log messages, and you can include system messages by reading and writing to system logs just by changing files, it allows you to put the logs together if you are using the whole computer like a vehicle's computer. Many people start off and build their own logging facility, divorced from the operating system. Now, if you have a specific justification for doing that, like you are operating cryptography or military systems and you don't want the enemy to guess the right system set up, then, that makes sense. It's a rational argument to keep your logging code separate. But for the rest, it creates EXACTLY the wrong kind of system development onus: you are now stuck maintaining something that really doesn't need to be included. The whole idea of the Linux/Unix kernel operating system is it abstracts away as much of the internal application programmer interface (API) so it can change and you don't have to worry (most times- kernels do change over epochs) about the internals. I have reviewed and fixed code working in the 1990's that crashes out of the tarball on recent linux kernels.

You get the most you need from written code you don't even need to rewrite, only learn, only reuse. That's the win-win you are looking for!

In real life, in a code base made by others, you will most likely find these three source concepts (daemons, forks, and file locks) somewhat written loosely together, or in 4 different files spread between one or more file folders. That made it easy to plan and execute three reuse operations: full extract from daemons/ into destination folder, full extract from filelock/ into destination folder, and partial extraction of header file lib/tlpi_header.h into destination folder.

I would write out what the operations you plan to use somewhere in the destination folder, I have written my explanation into the main.c file and the more importantly into the Makefile.am I added autoconf, libtool, and automake as I do because I prefer autotools to the clanky junk they offer. Now, autotools does have a nasty habit of caching things that you can't get rid of before you compile it properly, but the way around that is to do a make dist clean which destroys all configuration cached files. And then do these steps: autoreconf -fvi (removes and replaces all files needed for a fresh configuration), ./configure(find all configurations needs specified and outputs Makefile full of variables and recipes to make), make all(to complete all recipes to “make” whatever the recipes call for).

autoreconf -fvi (removes and replaces all files needed for a fresh configuration)
./configure
make all

That gets rid of most problems.

The reuse steps went as follows: from a file browser I created a folder called: Logger. Inside that folder I created a folder called originally originals. This isn’t how I do it myself in RULA, because there’s a bunch of extra stuff that’ll look like junk to anyone not familiar with a basic library setup first. It’s just a lot of junk if you aren’t familiar. This project is aimed at the crawl phase of learning to walk.

After the folders were created, I copied over all .c and .h header file from my version of Michael Kerrisk’s tlpi-XXX.tar.gz[10] (the directory from my linux home drive was : ~/src/tlpi-160429-dist/tlpi-dist/filelock/ the tilde ~ means home directory) folders called filelock and daemons (~/src/tlpi-160429-dist/tlpi-dist/daemons/) into originals. I copied over one Makefile from filelock/ for reference and I copied over the needed header file for all his common code in a folder called lib/tlpi_header.h. But I didn’t copy any of the other files from lib/* because I AM GOING TO USE THE GCC COMPILER TO TELL ME WHAT I AM MISSING BY CAUSING ERRORS. Some people work all day to avoid errors, in reuse the compiler is your best friend in explaining what you didn’t understand by reading the code. I don’t want to keep too much klag from Michael’s include folder, because I want this to stand alone as a logger without extra code. That’s my design choice. So I will bring over the minimum and then handle "missing" warnings and errors ("can’t find header XXX.h") to tell me what I need extra. Then, and if, I will SELECTIVELY ADD BITS to the Logger/tlpi_header.h (not the original but the copy) from lib/* into the header file. That chops off all needed dependencies.

It was at this point I read Mike’s Makefile.inc to realize his work needs all of lib/libtlpi.a - his library libtlpi.a (The Linux Programming Interface = tlpi) which is a static object because the library target is a "libXXX.a". So I copied that library folder lib/* as it’s own library for compilation. I would then, if I couldn’t extract - or if it would be too time consuming to transfer stuff out - just compile it all and add it to the target.

autoscan .

At this point, I ran an autotools tool called autoscan [1]("autoscan .") inside the originals folder with all the .c and .h headers. It does like it sounds, it reads all your source code files and creates a started configure.ac (autoconf file) that it defaults outputs to "autoscan.log". I then copied the autoscan.log file above into the Logger folder for use with configuration. And then, because I had to bring over the larger file lib/ I ran autoscan inside lib/ and compared the two inside the higher folder Logger. At this point, I realized Mike has symbolic links to code everywhere else inside his lib/ folder. So I ran autoscan in the properly configured original folder tlpi-dist/ to find the right stuff and then copied it over to the Logger folder.

With the two autoscan files, I was able to get a pretty comprehensive autoconfigure script file creation. From these two, I had the beginning of the complete configuration.

I decided to include libtool that handles library construction. I then made the simple decision to copy over the actual (originally symbolic linked) files from the separate library directories into the originals and then into the Logger folder for the reused library.

This is one of those hard versus easy path choices, the easy way is to keep everything the way it is until you have a reason not to. The library functions just fine as it is.

I then created a Makefile.am for automake configuration, a Makefile.in which is the intermediate makefile needed to create a final Makefile. Autotools demands you have some cruft files called AUTHORS, and so on it will tell you then you just call touch "filename" to create an empty file with the right name. I made a copy of the configure.ac in the top directory.

I created a library target recipe and I decided to copy over all the libtlpi.a files into originals and copied into Logger file folder.

In the top level file folder I created Makefile.am, Makefile.in, and Makefile by touching them. I copied the original autoscan.log back into the originals folder. I altered the configure.ac to check specifically for the syslog functions and commented on it in the configure.ac file.

There's an additional configuration step that isn't needed for this case but I included as I often do in case I expand what is here. You create a file called config.in and in the process of configuration inside configure.ac you add the following recipe: AC_CONFIG_HEADERS([config.h]) at the top of the file. It may or may not be automatically included depending on the version of your autotools. For recent linux like Fedora 28, it is automagically included.

[1] https://www.gnu.org/software/autoconf/manual/autoconf-2.67/html_node/autoscan-Invocation.html

I copied over some template library target recipes from other project Makefiles into the newly created Makefile.am. The other two files Makefile.in, and Makefile are automatically rewritten from the .am file.

Then I used a tool called meld - it allow you to examine 2 files in data or even 2 directories (or three items) to compare and contrast them. I looked at the Kerrisk lib folder versus the one higher directory originals. There were many more .c and .h files inside the library. Since I know the library works if you just compile all the pieces together (which is a big deal if you want it just to work from the start), I made the decision to copy over all missing files inside the originals/ folder from the available files inside the lib/ folder. Then I made the decision to copy all .c and .h files over into the top level directory Logger/ and build the library there. Then I would make a library of all the files save the one function called main.c.

Since we want a daemon logger at the end, and a library supporting that main file. Why? Because the convention (remember formalisms as a pain of misunderstanding?) is the *de facto* default C main.c source file for an executable program. If you want people to understand it's easier to use the *de rigeur* conventions unless there's a compelling reason not to.

So all original source files are within the originals/ as is. Now I copied over all the .c and .h files into the top level folder Logger as is. No changes are made to the originals. Inside the originals directory:

```
cp *.h *.c ..
```

Now I need to find the main and isolate it in main.c and finish setting up configure.ac with the right modifications, and decide how to set up the automake Makefile.am file as to the targets a library and a main.c executable daemon program.

I loaded files from the Logger into my preferred text editor geany (I used to love Source Commander until they broke it and left it). Then I use the search files function to find any mains main.c inside all the source files. Remember, I haven't read the files yet. The majority of work so far is just treating the file objects as working if they all go together without any specific knowledge. Using the command:

```
find "main("
```

(because all main functions must have this specific text I find (geany uses grep so the command line command was):

```
grep -nHIF -- "main(" *
    daemon_SIGHUP.c:126:
    main(int argc, char *argv[]) error_functions.h:28: terminate
main() or some other non-void function. */
    i_fcntl_locking.c:96:main(int argc, char *argv[]) grep: originals:
Is a directory test_become_daemon.c:21:main(int argc, char *argv[])
t_flock.c:23:main(int argc, char *argv[]) t_syslog.c:33:main(int argc,
char *argv[])
```

And this is valuable for two reasons: one, at the end we want all the functionality to start from one main() function, and two, we want some of the functionality from these separate test examples included in the same main.c file.

We want these things in the final version:

1. We want a daemon process started as expected (a forked child that runs and exits after forking a second child with no terminal I/O) as Michael Kerrisk explains in his Chapter 37[10]: Daemons[10].
2. We want an interrupt driver for the kernel SIGHUP (SIGNAL HANG UP) signal to reload configuration files.
3. We want a locking file as described in Chapter 55 (Listing 55-1)[10] to make sure there's only one daemon controlling access to log files.
4. We want a properly configured configuration file with explanations populated into the Logger project.
5. We want the log file to be readily apparent in the literature including the README file.
6. We want basic instructions included in the README file to explain all this.
7. We want an INSTALL file to explain in numerical sequence what commands to call to make this project work.

Now I went through the process of just adding the non main.c files to a library and getting it running. Now, and this is where many people have problems and give up, the one major downside to Linux programming is the sheer scale of software packages, including in this case what Fedora calls "XXXXX-devel" development .rpm (Red Hat Package Manager) packages. If you haven't installed GCC C compiler, GCC C++ compiler, GNU Autotools autoconf, automake, autoreconf, libtool, and helpful software tools like Geany (text editor), and Meld then you will run into all the problems.

All these problems require you to know what the NAME of the package is called so you can download and install the right ones. This is a reality of using this system and I can ASSURE YOU it was FAR FAR WORSE in the earlier days of Linux programming. You have in Yum, RPM, and in the Ubuntu apt-get tools of sophistication that will find all the dependencies if you have the repositories set up right. But there is no shortage of teething pains getting a system working. I have learned over the years to get the programming software stable on a distribution and then cease all upgrades so I know it's solid.

What I can do, is provide you with a DVD release with all the entire software rpm's needed to get you a properly configured FIRST distribution with all the tools. It costs $5 USD, it's not my code, it's still Fedora and under their license rules but it will save you time and frustration for $5. I've spent entire weeks getting a safe working software version including working OpenGL extensions. The DETAILS are listed in the test file SAFE_PROGRAMMING_DISTRIBUTION inside the RULA and the Logger project folder.

I set to work, after a nap, fulfilling the configuration of the autotools by copying over some configure commands from another autoconf/automake/libtool project folder to make the GNU Autotools configuration complete. The file configure.ac has all the inclusions commented. This was mainly cut and paste them overly wide comments for you. Over time, you will comment the obvious less and less.

Then I ran:

autoreconf -fvi

to force autotools to create a new project here -f forcing to include files, -v being verbose so I could see the messages, and -i including instead of symbolic linking to other files. It did things like setup automake and libtool as you can see in the verbose messages:

autoreconf -fvi

```
[dave@localhost Logger]$ autoreconf -fvi autoreconf: Entering directory '.'
autoreconf: configure.ac: not using Gettext autoreconf: running:
aclocal --force aclocal: warning: couldn't open directory 'm4': No such file or directory
autoreconf:
configure.ac: tracing
autoreconf: running: libtoolize --copy --force
libtoolize: putting auxiliary files in '.'.
libtoolize: copying file './ltmain.sh'
libtoolize: putting macros in AC_CONFIG_MACRO_DIRS, 'm4'.
libtoolize: copying file 'm4/libtool.m4'
libtoolize: copying file 'm4/ltoptions.m4'
libtoolize: copying file 'm4/ltsugar.m4'
libtoolize: copying file 'm4/ltversion.m4'
libtoolize: copying file 'm4/lt~obsolete.m4'
libtoolize: Consider adding '-I m4' to ACLOCAL_AMFLAGS in Makefile.am.
autoreconf: running: /usr/bin/autoconf --force
autoreconf: running: /usr/bin/autoheader --force
autoreconf: running:
automake --add-missing --copy --force-missing
```

```
configure.ac:22: installing ’./compile’
configure.ac:15: installing ’./config.guess’
configure.ac:15: installing ’./config.sub’
configure.ac:19: installing ’./install-sh’
configure.ac:19: installing ’./missing’
Makefile.am: error: required file ’./NEWS’ not found
Makefile.am: error: required file ’./AUTHORS’ not found
Makefile.am: error: required file ’./ChangeLog’ not found
Makefile.am: installing ’./COPYING’ using GNU General Public License v3 file
Makefile.am: Consider adding the COPYING file to the version control system
Makefile.am: for your code, to avoid questions about which license your project uses
autoreconf: automake failed with exit status: 1
```

It failed. Exit status:1 means failed to make proper configure shell script to make a project. This tells us what needs to be fixed in the error: commands above.

The program autoreconf does an "autoreconfiguration" from before the working files are made. Those working files like configure.guess, configure, and compile among others do the work for your tools with custom configuration for everything inside this folder. The whole story of GNU autotools[2] takes a large book so just accept this from now.

This reminded me what the extra files I need to add are. So I ran this command:

touch NEWS AUTHORS ChangeLog COPYING

This made autoreconf happy and it successfully created everything:

autoreconf -fvi
libtoolize
configure.ac
configure
automake
make
ld
AC_OUTPUT

```
[dave@localhost Logger]$ autoreconf -fvi
autoreconf: Entering directory ‘.’
autoreconf:
configure.ac: not using Gettext autoreconf: running: aclocal --force autoreconf:
configure.ac: tracing autoreconf:
running: libtoolize --copy --force libtoolize: putting auxiliary files in ’.’.
libtoolize: copying file ’./ltmain.sh’
libtoolize: putting macros in AC_CONFIG_MACRO_DIRS, ’m4’.
libtoolize: copying file ’m4/libtool.m4’
libtoolize: copying file ’m4/ltoptions.m4’
libtoolize: copying file ’m4/ltsugar.m4’
libtoolize: copying file ’m4/ltversion.m4’
libtoolize: copying file ’m4/lt~obsolete.m4’
```

[2] A really good unofficial blog on GNU Autotools confusings is here: https://autotools.info/

```
libtoolize: Consider adding '-I m4' to ACLOCAL_AMFLAGS in Makefile.am.
autoreconf: running: /usr/bin/autoconf --force
autoreconf: running: /usr/bin/autoheader --force
autoreconf: running: automake --add-missing --copy --force-missing
configure.ac:22: installing './compile'
configure.ac:19: installing './missing' autoreconf: Leaving directory '.
```

This output tells you the Autotools ARE CONFIGURED FOR USE. If there are any errors that appear it is YOUR CONFIGURE.AC and AUTOMAKE FILE that is the cause. Not the background tools. You see libtoolize messages with no errors so libtool is set up. You see autoconf messages meaning ./configure is ready to configure. This DOESN'T MEAN your configure.ac is correct, it means the errors you see with command ./configure (ALWAYS RUN WITH ./ before so you are using THIS configure, not the first configure it finds in any path!) are due to your commands. You see an autoheader message and that means your config.h header is ready to be configured. It also created a missing folder called m4/ because that's where it stores local macros used by the m4 language to make the configurations underneath the ./configure command.

In order to get this project to work, we need to make a library inside Makefile.am in commands automake understands, and then make sure configure.ac looks for any dependencies that configure will need to make sure autoconf and automake will work as needed.

Since we are going to do everything in this folder, there is no subdirectory command in configure.ac to go looking in originals. There is also just one AC_OUTPUT target called Makefile inside the last command (IT MUST ALWAYS BE LAST!) AC_OUTPUT([]). AC_OUTPUT does the work of making Makefile from Makefile.in which holds the intermediate macro values from operating the Makefile.am commands.

Now, before I did any code manipulation, I read the COPYING file autoincluded by autoreconf. Then I read the original COPYING file from Michael Kerrisk's software distribution. They were the SAME GPL - GNU Public License V3 versions. So there is no conflict with adding, modifying and distributing.

What I did next is add a COPYRIGHT DISCLAIMER comment into EVERY SOURCE FILE in Logger.

COPYRIGHT DISCLAIMER

It reads:

```
/*! \file **************************************************************\
* \copyright { Copyright (C) Michael Kerrisk, 2016. } * * *
* This software below is modified from source tlpi-160429-dist.tar.gz *
* This software is governed by the same terms as laid out below. *
\***********************************************************************/
```

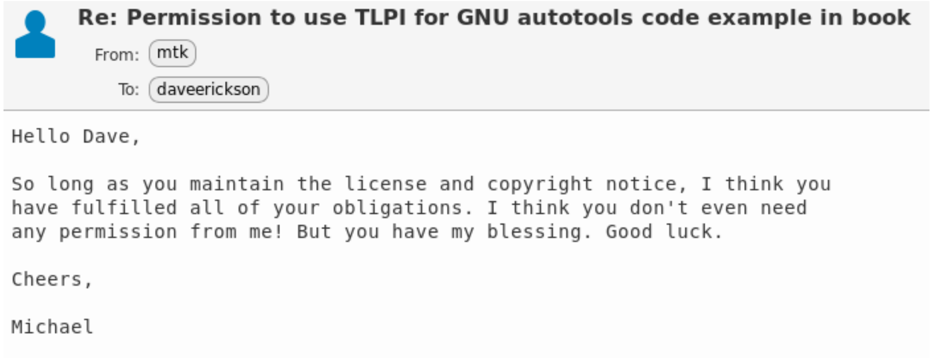
A disclaimer is a term and condition that notifies others - the people reading the code - that you are not usurping - you are not in contravention - of the copyright of the files in this project. You are claiming fair use, as designated either by the terms and conditions set out or under the accepted considerations of fair use (we cover this later in 9). This disclaimer claims you are not claiming the code. They are still Michael's code files, you are merely modifying them for your use and letting everyone know. But also note you AREN'T adding these copyright disclaimers to YOUR Autotools files. He never had Autotools, you are adding it. It's not his to worry about.

\copyright
doxygen

Now, there are changes in the disclaimer: one, I add a doxygen \file name command to the top row for documentation, two, I add a \copyright so doxygen gets the copyright AS MICHAEL wrote it (No interpretation, verbatim), a line pointing to the underlying terms, and the exact source tarball I took it from. You do this once, copy it into all files at the top above the original copyright notice. You haven't changed his terms, but you make it clear that anything under may not be as written. I will change one of the main() functions into the main.c files but the code is identical so I set it to the same terms. You do this once for all code and you add his copyright terms and conditions and you're in compliance because that's what those terms state.

It took about 10 minutes to copy that header onto the type of file. You haven't stolen anything, you haven't taken on any added accountability. I copied over Michael's COPYING files as he mentions into this file folder. Legal part satisfied. That was 56 files:

alt_functions.c alt_functions.h become_daemon.c become_daemon.h binary_sems.c binary_sems.h create_pid_file.c create_pid_file.h curr_time.c curr_time.h daemon_SIGHUP.c error_functions.c error_functions.h event_flags.c event_flags.h file_perms.c file_perms.h get_num.c get_num.h i_fcntl_locking.c inet_sockets.c inet_sockets.h itimerspec_from_str.c itimerspec_from_str.h print_rlimit.c print_rlimit.h print_rusage.c print_rusage.h print_wait_status.c print_wait_status.h pty_fork.c pty_fork.h pty_master_open.c pty_master_open.h rdwrn.c rdwrn.h read_line_buf.c read_line_buf.h read_line.c read_line.h region_locking.c region_locking.h semun.h signal.c signal_functions.c signal_functions.h test_become_daemon.c t_flock.c tlpi_hdr.h t_syslog.c tty_functions.c tty_functions.h ugid_functions.c ugid_functions.h unix_sockets.c unix_sockets.h



Re: Permission to use TLPI for GNU autotools code example in book

From: mtk

To: daveerickson

```
Hello Dave,

So long as you maintain the license and copyright notice, I think you
have fulfilled all of your obligations. I think you don't even need
any permission from me! But you have my blessing. Good luck.

Cheers,

Michael
```

Michael Kerrisk's gracious blessing...

So when people complain that legal stuff is a burden, you can show them as long as the code terms and conditions are very flexible like (lesser) GPL Licences it takes little time at all.

In fact, I sent an email to Michael requesting his permission to use his code just to be sure. Given the topic of my book I wanted to make certain for obvious reasons. And like the code boss he is, in Figure 14.1, this was his response:

Makefile.am

Back to code.

Here is the made up Makefile.am in the Logger/ folder describing to autotools to make an executable program called LoggerD and a supporting libtool library (which can make static .a and dynamic .so shared object libraries) to make Michael's code into libtlpi.la which is a special libtool library. You need only provide different flags to make a static or shared object library.

# Comment

```
# DRE 2022 The executable program LoggerD - Yo Yo Yo
```

bin_PROGRAMS

```
bin_PROGRAMS = LoggerD
# DRE 2022 this command tells libtool to build a library called lib_LTLIBRARIES = libtlpi.la
# library's source files - added one per line but no headers and no main() files:
libtlpi_la_SOURCES = alt_functions.c\
become_daemon.c \
binary_sems.c \
create_pid_file.c \
curr_time.c \
error_functions.c \
event_flags.c \
file_perms.c \
get_num.c \
inet_sockets.c \
itimerspec_from_str.c \
print_rlimit.c \
print_rusage.c \
print_wait_status.c \
pty_fork.c \
pty_master_open.c \
rdwrn.c \
read_line_buf.c \
read_line.c \
region_locking.c \
signal.c \
signal_functions.c \
tty_functions.c \
ugid_functions.c \
unix_sockets.c
# ENDS WITH CARRIAGE RETURN LAST LINE
# DRE 2022 - Libraries LIBADD (Libraries ADD) while programs LDADD which means linker daemon (ld) ADD
libtlpi_la_LIBADD = -lc
# I added libc just because but it’s already added
```

I excluded the four files for tests that included a main() function, as they can all be make check test files of various functionality they were made for. I will then include them into Makefile.am as check test programs.

Here are the new make check tests out of the four existing files added to the Makefile.am in the LoggerD new folder:

_SOURCES =
_CFLAGS =
_LDFLAGS =
_LDADD =

```
# DRE 2022: test programs for the library straight from Michael Kerrisk’s examples:
check_PROGRAMS = test-daemon test-syslog test-flock test-fcntl-locking
# DRE 2022: Become a Daemon check target
test_daemon_SOURCES = test_become_daemon.c
test_daemon_CFLAGS =
test_daemon_LDFLAGS = ${LOGGER_LFLAGS}
# DRE 2022 the added The Linux Programming Interface library in .la format.
test_daemon_LDADD = -lm -lc libtlpi.la ${LOGGER_LIBS}
# DRE 2022: Become a Daemon check target - I changed .c filename to a longer more clear name
test_syslog_SOURCES = test_syslog.c test_syslog_CFLAGS = test_syslog_LDFLAGS = ${LOGGER_LFLAGS}
# DRE 2022 the added The Linux Programming Interface library in .la format.
test_syslog_LDADD = -lm -lc libtlpi.la ${LOGGER_LIBS}
# DRE 2022: Become a Daemon check target
test_flock_SOURCES = test_flock.c test_flock_CFLAGS =
test_flock_LDFLAGS = ${LOGGER_LFLAGS}
# DRE 2022 the added The Linux Programming Interface library in .la format.
test_flock_LDADD = -lm -lc libtlpi.la ${LOGGER_LIBS}
# DRE 2022: Become a Daemon check target
test_fcntl_locking_SOURCES = test_fcntl_locking.c
test_fcntl_locking_CFLAGS =
test_fcntl_locking_LDFLAGS = ${LOGGER_LFLAGS}
# DRE 2022 the added The Linux Programming Interface library in .la format.
test_fcntl_locking_LDADD = -lm -lc libtlpi.la ${LOGGER_LIBS}
```

This allows us to use the verbatim testing of Michael's own test main() files as is.

Once the library is made, now we have 4 ways to test the functionality inside the library. Note, there are more function files than these code snippets needed. But we won't fix the rest unless it's needed.

There is an important factor, we haven't even read the code yet. I will attempt to make the library work as expected first. Now, as I start compiling and debugging, I realize the only way the code won't work is if there is nonconfigured items that source is looking for. Because the code is identical.

So I opened the originals/Makefile.inc to find his generic include CVARS that his compiles expect from that I see a bunch of libraries that I should blanket include to all compiled targets:

Here's the original:

```
LINUX_LIBRT = -lrt
LINUX_LIBDL = -ldl
LINUX_LIBACL = -lacl
LINUX_LIBCRYPT = -lcrypt
LINUX_LIBCAP = -lcap
```

`Here’s␣my␣updated␣configuration␣inside␣Makefile.am:`

_LIBS =

```
#DRE 2022: Added for Linux libraries
LOGGER_LIBS = -lrt -ldl -lacl -lcrypt -lcap
```

I added this FLAG ${LOGGER_LIBS} that is reinserted by configure when it runs configuration into all the targets inside Makefile.am.

Now with these needed libraries, I included them into the configure.ac as additional checks for stuff.

For the librt.so I located the file inside the /usr/lib64 folder, so I added the LDFLAGS flag -L/usr/lib64 to the Makefile:

```
#DRE 2022 LOGGER_LFLAGS = -L/usr/lib64
```

I then used nm to name the objects inside the librt.so object to find one to look for: I found clock_settime as a function and added it to the configuration:

nm /usr/lib64/librt.so
AC_CHECK_LIB

```
AC_CHECK_LIB([rt], [clock_settime])
```

I did the same for the rest:

```
AC_CHECK_LIB([rt], [clock_settime])
AC_CHECK_LIB([dl], [dlmopen])
AC_CHECK_LIB([acl], [acl_init])
AC_CHECK_LIB([crypt], [crypt_r])
AC_CHECK_LIB([cap], [cap_size])
```

By using a selected member of the libraries the autoconfigure will succeed at finding and testing them.

I won't worry about the include headers because all Linux glibc include headers are added by default into the default /usr/include folder. But those could be added.

I started to autoreconf and it worked with some slight changes.

I ran ./configure and it failed because I hadn't changed the command:
AC_CONFIG_SRCDIR

```
AC_CONFIG_SRCDIR([syslim/t_sysconf.c])
```

to include a source code from the current folder.

```
AC_CONFIG_SRCDIR([print_rlimit.c])
```

Then I reconfigured and ./configure worked on the SECOND TRY.

If you get errors on any configure, it MEANS YOU ARE MISSING DEVEL FILES AND/OR LIBRARIES for glibc because that's what GNU gcc uses. You may need to install more libraries and things like libacl-devel development libraries and include headers.

This will happen, I can't stop and go through the myriad of possibilities, you can email or insert a ticket in the gitlab and I may help you out.

What this means is your configuration for this library and executable program is finished for now.

I ran the first make of the library:
make libtlpi.la

and this was the output:

```
    ... functions.Tpo -c error_functions.c -fPIC -DPIC -o .libs/error_functions.o error_functions.c:26:10: fatal
error: ename.c.inc:
    No such file or directory #include "ename.c.inc"
    /* Defines ename and MAX_ENAME */ ^~~~~~~~~~~~~ compilation terminated. make: ***
[Makefile:751: error_functions.lo] Error 1
```

I forgot an additional file, ename.c.inc; i then included in the Logger/ folder.

I got all the files to compile with no CFLAGs set, I don't use that first time and only insert it if code has a comparability problem. Since the code is from one coder and one project, that won't be an issue here.

All the files then compiled, I did see some warnings as they flew past (this machine has 8 CPU's - I recompile ATLAS with "make -j 8" and it looks like artillery fire) but I won't do anything until there's a seg fault crash. But if you see warnings go by it might be a problem you will need to debug later.

On the third compile try, it compiled the library.

```
[dave@localhost␣Logger]$␣make␣libtlpi.la
```

```
    libtool: link: gcc -shared -fPIC -DPIC .libs/alt_functions.o .libs/become_daemon.o .libs/binary_sems.o
.libs/create_pid_file.o .libs/curr_time.o .libs/error_functions.o .libs/event_flags.o .libs/file_perms.o
.libs/get_num.o .libs/inet_sockets.o .libs/itimerspec_from_str.o .libs/print_rlimit.o .libs/print_rusage.o
.libs/print_wait_status.o .libs/pty_fork.o .libs/pty_master_open.o .libs/rdwrn.o .libs/read_line_buf.o
.libs/read_line.o .libs/region_locking.o .libs/signal.o .libs/signal_functions.o .libs/tty_functions.o
.libs/ugid_functions.o .libs/unix_sockets.o -L/usr/lib64 -lcap -lcrypt -lacl -ldl -lrt -lc -g -O2 -Wl,-soname
-Wl,libtlpi.so.0 -o .libs/libtlpi.so.0.0.0 libtool: link: (cd ".libs" && rm -f "libtlpi.so.0" && ln -s "libtlpi.so.0.0.0"
"libtlpi.so.0")
    libtool: link: (cd ".libs" && rm -f "libtlpi.so" && ln -s "libtlpi.so.0.0.0" "libtlpi.so") libtool:
link: ar cru .libs/libtlpi.a alt_functions.o become_daemon.o binary_sems.o create_pid_file.o curr_time.o
error_functions.o event_flags.o file_perms.o get_num.o inet_sockets.o itimerspec_from_str.o print_rlimit.o
print_rusage.o print_wait_status.o pty_fork.o pty_master_open.o rdwrn.o read_line_buf.o read_line.o re-
gion_locking.o signal.o signal_functions.o tty_functions.o ugid_functions.o unix_sockets.o libtool: link: ranlib
.libs/libtlpi.a libtool: link: ( cd ".libs" && rm -f "libtlpi.la" && ln -s "../libtlpi.la" "libtlpi.la" )
```

Success. So I tried the first test first in the make checks: make test-daemon

```
    [dave@localhost Logger]$ make test-daemon
    gcc -DHAVE_CONFIG_H -I. -g -O2 -MT test_daemon-test_become_daemon.o -MD -MP -
MF .deps/test_daemon-test_become_daemon.Tpo -c -o test_daemon-test_become_daemon.o ‘test -f
’test_become_daemon.c’ || echo ’./’‘test_become_daemon.c mv -f .deps/test_daemon-test_become_daemon.Tpo
.deps/test_daemon-test_become_daemon.Po /bin/sh ./libtool --tag=CC --mode=link gcc -g -O2 -L/usr/lib64 -o
test-daemon test_daemon-test_become_daemon.o -lm -lc libtlpi.la -lrt -ldl -lacl -lcrypt -lcap -lcap -lcrypt -lacl -ldl
-lrt -lc -lc -lc -lc libtool: link: gcc -g -O2 -o .libs/test-daemon test_daemon-test_become_daemon.o -L/usr/lib64
-lm ./.libs/libtlpi.so -lcap -lcrypt -lacl -ldl -lrt -lc -Wl,-rpath -Wl,/usr/local/lib
```

It compiled.

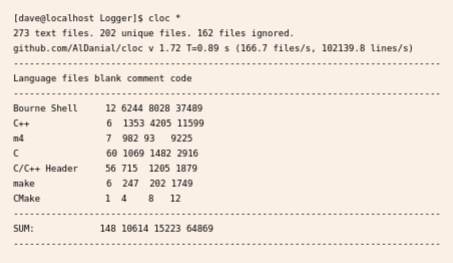
I added a subdirectory to the project in astyle, which I found in the BRL-CAD code as an addon tool and installed within. After I added the C++ compiler check called AC_PROG_CXX to the configure.ac file, it configured astyle/ correctly and compiled on the first try. With no other additions, this project compiles with GNU g++ C++ compiler (see below). Astyle formats and code to a file proforma format as given in a configuration file.

AC_PROG_CXX

```
# DRE 2022 - added CPP C++ compiler
AC_PROG_CXX
```

In sum, here are the off-the-top performance metrics of this reuse project (using the program cloc):

cloc



```
[dave@localhost Logger]$ cloc *
273 text files. 202 unique files. 162 files ignored.
github.com/AlDanial/cloc v 1.72 T=0.89 s (166.7 files/s, 102139.8 lines/s)
-------------------------------------------------------------------------------
Language files blank comment code
-------------------------------------------------------------------------------
Bourne Shell     12 6244 8028 37489
C++              6  1353 4205 11599
m4               7  982 93   9225
C                60 1069 1482 2916
C/C++ Header     56 715  1205 1879
make             6  247  202 1749
CMake            1  4    8   12
-------------------------------------------------------------------------------
SUM:            148 10614 15223 64869
-------------------------------------------------------------------------------
```

In about 10 man hours, going much slower than I normally do because I was stopping to write into the book, step by step, and with ample breaks and naps, writing extra documentation, I reused 25619 LOC lines of code of source code (all C, C++ and m4 source code) and 64869 in all languages. The work wasn't strenuous, and I went slowly to plan what I was doing. I must bring forward the factor that because this code was intended for a Linux machine and all the samples were working this was a very easy reuse. I needed to add two libraries and include headers, the ability to understand which include headers is an added skill that comes with practice, to get it all to work. And I used these commands/tools: GNU autotools (autoconf, automake, aclocal, libtool), meld, cloc, nm, locate, which, yum install, gcc, g++, GNU ld, GNU make, grep, and Geany. I won't add the time making a LoggerD daemon final product, I am just going to get it working and you can read it in the repository. But in fact, that won't be much more, I am going to just copy bits of the main functions into a working one. That's it. I didn't know it would work this easy but using my experience and the proscriptive tips from above in this book it was easy.

If you use the very low cost per LOC value of $18USD/LOC, then you have an estimated value of ~$461,142.00USD. Investing 10 hours work to exploit existing code converted to over $400,000 value reused / leveraged for new projects. If you divide that over many projects for a logging function used by many software projects, which was the goal, and you have added that value to ALL of them. Even if you restrict the code leverage down to just C and C++, then the reused LOC is 16394, and the reused value is $295,092USD. Neither is not too shabby. Imagine the value at $50USD/LOC. Over a $1 million USD for 10 hours work.

Some might say, well hang on, you didn't include that extra LoggerD function, and that's true. But I am a perfectionist when it comes to code and I will naturally expend more time and effort than most people rewriting code that is ready for expansion and not just working. Most of that won't be captured as extra value because I will try a few varieties to see what I like better. In this way, my pedantic nature would skew the results more than most people would agree with. In other words, if I rewrite 300 lines of code 5 times, most people wouldn't count that as 1500 LOC, so my numbers would start to look off. I tried to make as objective an exemplar project for the purpose of proving my words. I believe adding that extra would muddy the waters and not in a productive way. Going into the Logger Reuse Code Exemplar I had no idea how it would turn out nor how long it would take. I was only confident based on years of failing that I had the right skill set to overcome any problem. And I knew Michael's code was straightforward and sound. And you will notice above all I looked at very little code to understand how to fix it. So it is very likely that the better you are at the reuse strategy skill, the faster you can extract value.

In fact, I posit that by not getting bogged down inside the code is the very way you save time and increase value. And this honest example proves it.

If there is any doubt in your mind you can extract value from code, it takes little more than what's here. You will struggle in your conversions. But bringing a methodical approach, writing out your approach, your code reuse strategy, and making all steps clear you will leverage software.

Chapter 15

Reuse Legal Actions

Let The Past Not Make You Bitter,
But Better.
—Siddhartha Gautama, The Buddha

Legal Aspects of Code: The Good, The Bad, and The Nonavailable

While contract law and software patents are known within many Western legal systems, the most relevant problem you face may have to do with abandoned software.

I weigh in on the issue from a couple of ways, and I parallel the code with other precedented articles of property to put what that problem may actually mean.

The most important factor to any contract attached to the code you review is what the contract actually says. You need to become familiar with the terms and conditions, or get someone to describe what they should mean to you.

But here is the problem writ large: you write one line of code, label it and sign it as your property and declare the legal terms and conditions, you have created something of property under copyright, under trade secrets, perhaps under trademark, and in some nations under software patents. You staple a disclaimer to it and an indemnification about any liability that the code may get used, and you have a legal product that must be respected by anyone wanting to use it.

Here's an analog of this software issue: you hold a class at the local college and you want the students to have a copy of just a half chapter of a really good but expensive book. You don't want already stretched students eating amen for dinner to pay for a complete book. Under the rules of fair use, you can photocopy a "few" pages of a copyrighted book and distribute them so long as it is not for commercial purposes. As long as you acknowledge the originator you can photocopy it. This has been established. So there is a gray area on the rights and fair use.

Consider the case of a piece of artwork. If a painting is a set of painted blotches on top of a simple cloth canvas on a crude wooden frame, then what is the intellectual property? Well, if someone photographs that painting, it's a breach of the original property even though the cost of an digital image is near zero. It's a simile, and even a facsimile. Clearly, the value is inside the way the paint blotches are arranged on the canvas. But the value inside a software source code file can't be the protected words of the language syntax, can't be your name and can't be obvious words that are in the public domain. So even if you "copied" a section of code the actual value, especially if the software was abandoned, can't be infinite. And even if the code was written and terms clear, how can it have any value to the owner if abandoned in an old university webpage or a dormant software repository? Isn't garbage worthless?

Here's another problem with where the valuable parts of code are. Even if you never copy a line of code from one open source code file into another, you will be (probably) extracting as much of the value out of it as you might have just reading and understanding it. Every programmer writes names of variables and assigns them a nomenclature identity. Every one makes a main function to start things off. You aren't learning any thing new about initializing variables from any one's code that you would have already known from your education. When you read code, and I have read over 5000 different projects, you understand the interal data flow and the math algorithms being used. For me, I look at the source directory, scan the file names, understand the folder layout and I have gained a cursory but high-level view of what the code does. I don't even need to read some files to see how the software is organized. That's about it. If there is something particulary insightful or a formalism to present the code is nonstandard, in most cases the proper older syntax for the same code you learned in a later version, there is something you don't expect, so it's novel. But that's it. Most math is in the public domain, many websites explain how to do math. Unusual things like how many iterations do they use to calibrate a Kalman filter, or what kind of data sets do they store to relearn a machine learning function running in real time with new data, these are the kinds of things you see in code that you won't necessarily be aware of. Another big one is the proper data formatting for the operation of functions inside other code like the proper way to feed matrices to ATLAS (Automatically Tuned Linear Algebra Software), or a more common one is how do you send a vertex array and normals array to an OpenGL graphics function to get nice output and not garbage. What are the proper variables in the manner a function expects. That's what people struggle with when getting other code to work and that is valuable to others that haven't figured out how to do that yet. But other than that, it's all *proforma*. Mainly methodical boilerplate code. The cost of typing for a lot of it will be far less than people imagine.

Now, the main parts of the value will also be derived from the testing, compiling onto a custom CPU or MCU, quality assurance, research, development, meetings, marketing and other efforts like deployed operation, that come with non-recurrent costs like feeding and housing a design team on a customer's on-site factory, and so on.

The next exemplar is a treasure chest of gold at the bottom of an ocean. To be clear, the gold coins may be emblazoned with the late King Ferdinand's bust as the property of Spain. But centuries after his death and in the unclaimed neutral waters, part of a sea, with no descendants actively attempting to recover the treasure (descendants that may not even know the treasure was lost) those lost coins are subject to the law of the sea known as The Law of Finds. In the case of a shipwreck that the owner has given up all hope of salvage, the discoverer is entitled to 100% of the value found within. The discoverer has been deemed to have full rights to any salvage. Clearly, the treasure of a software effort, lost in the flotsam and jetsam of the public internet archives, is like a parallel to a disowned software as a pile of junk on the ocean floor.

Turn back to software with legal terms and conditions. What happens when you take (download) software and attempt to get it working while still attempting to obey the terms and conditions as laid out, when you post improved code back to the internet and no response is given? What if you get no reply back from the original email, perhaps it returns an error message as undeliverable, what if you have no way of making contact with any of the original authors (I use authors because I generalize copyright to be the most relevant law that is close to this cirmcumstance)? Are you duty bound to stop all forward progress? If you asked a judge or lawyer my guess is that they would advise you not to continue.

I argue the opposite. In fact, from personal experience dealing with a large array of developers (the good, the bad, and the nonavailable) that while the terms say one thing, the actions of the owners indicate the opposite. In this book, I sent out no less than 18 requests by email for permission to use various works, even if I probably didn't "need to". Within three months, I got most replies back, three missing in action, most affirmative and one negative but I got most responses in a reasonable timeliness, nonetheless. I argue that even if the software is within the legal terms and conditions of restricted copyright but the owner has abandoned the work with no response within 3 months, then that is more like a hub cap lost on a highway than it is a copyright property. It's public salvage or a treasure of the law of the finds. I have worked with more code than most people, and far more than any lawyer or judge will in their lifetimes. I know very few software projects live long enough to make a significant value, and thereby value lost for harm. And it would be very hard to predict and forecast what value might have been lost when the original code was in the public and abandoned for years.

When dealing with stuff, we normally don't write our name, address, and email on our underwear, with obvious exception of Sheldon Cooper. If a hub cap falls off your car and you don't see when and where it was lost, it's not like the person that picks it up and sells it for scrap doesn't know this is STILL OWNED by the owner. Like the Law of the Sea, and salvage laws, there must be an acceptance that lost software with missing owners should be treated like a hub cap without a name. You make one attempt to get through, and barring strenuous objections in a reasonable time, you should consider it like found salvage. Treat it like you would anything lost, of value or not.

All that being said, what I want the lawyers and judges foaming at the mouth at my scurrilous indictment of the value of legal contracts (well deserved in my case) to understand is the wider and more important societal implications here must favour a pro-salvage mindset for the long-off benefits on society rather than the short term and myopic interests of a lost owner. All software writers in the West were educated how through a public paid for public education system by taxpayers. The topics in code are neither unique to one person nor unique to any proprietary interest. I can reproduce similar code with enough false starts and practice runs without reading the original, and in my case I will probably exploit other's code instead of unique bits made in a panic to make my version work better, faster, less energy-dependent than the original programmer. There can be many copies of code that are similar and function almost identically with original software developed in isolation, that is a function of how computers compile software, NOT THE DEVELOPER. Compilers can optimize for size or speed so the code gets recreated by compilers now, there's a chance a compiler version makes an identical copy to an existing copyright by accident. Names are not important because compilers MANGLE names to make them hard to find in textual structure within machine assembly language to avoid software insecurity. For example, if someone can trap the software and run it, you as a user or customer don't want the hackers or thieves to find a function called "ENCRYPT_SOFTWARE_MESSAGE" in the object tables so they can target their attacks at messages. So, in this case, technically, there is a very limited span of uniqueness attributed solely to any one writer and thereto his/her/it's purported value.

For most cases, if you read the terms, you make one unsuccessful attempt to contact owners which you record, and then you can act in the best ways of humanity that pays for software. Record the attempt.

In most cases of my reading someone's code and reviewing it for my purposes, I use doxygen on the code base, I adjust the configuration file and adapt the make files to include a "make dox" build command that autogenerates the documentation back. Once I am done this, I attempt to send back my fixes as files on offer to the owner's for their consideration. And then most times I don't even use the code, or I use the "ideas" I learned from reading it without facsimile. Or whatever. And I have gotten maybe one reply out of hundreds of sent emails.

I keep copies of the original tarballs / zipped file folders. I keep the doxygen part handy in case they ask for it. I tarball a newer version if I add or improve some code. And I've never heard back asking for the improved code. A lot of developers DON'T want your fixed code, because it would force them to go backwards and figure out how to make the changes retroactive to the work they are adding. It causes them MORE WORK. This is an important legal point when considering what “harm” may have been done: in most cases there isn't a desire for code modifications that vary from the lead programmers desire. Even if you give them the diff and patch files (another GNU tools command set: diff charts out file changes and patch applies the difference into an original file turning it into an updated file) they would have to apply the patches and test them out. This is a very strong reason why most people fork (create a new repository with the same code at some point) to make their own version.

If you don't plan to commercialize, if you are using the software for your own purposes or for education, and so on the you don't have much to worry about. You aren't making a valuable thing someone will claim is worth a royalty.

At this point in your commercialization, you can guarantee that the limit of the code snippets and can prepare a substitution plan if needed. If you ever get served with a cease and desist letter, you have a plan to reverse anything that's demanded. If you get a cease and desist letter you also have a recent contact with which to communicate. Perhaps that's enough to make those parties feel included in your thoughts. If you have as I advocate herein to improve code, you should welcome this attempt as a long lost relative found overseas. Remember that code review to measure what the impact on risk exposure was, you can now promptly and politely describe precisely how little the impact is, no matter how small, of the code's improvement, you include your recorded contact attempt, and a plan to remove any code in your longer term code project management. Of course if they are that concerned about removing the code that would contact you and demand it, they must also be in a position to pay for the privilege of removing and replacing their code faster than scheduled. You can present them with an invoice to cover the costs. And, if you have estimated the cost of the controversial code, you can contrast that limited value with the overall investment in the non-recurrent engineering of the software code base in it's entirety. With your extensive software code review, you can show snippets of the original code and the updated, improved code snippets that show greater value - from better documentation to bug fixes - and that puts anyone complaining about unfair treatment that you are the one that's hard to dissuade anyone, including a reasonable judge, that your behaviour is above board and honourable.

Rather than worry about a disgruntled lone creator software entanglement, keeping in mind the longer goals, commit to the longer view by reviewing what needs to be replaced if a problem occurs, have a way to offroad that code, and even offramp all your software development. Think on it this way, why wouldn't you get ready or just get ready to sell your software assets to a potential investor/ company. You would prepare to excise and replace anything you needed in order to satisfy the due diligence of a company merger and acquisition team expects you to do. It is just another consideration you must manage with all the other problematic considerations: licence issues, bugs, internationalization, data privacy laws, and so on. These are all valid reasons why one should stop progress and prepare to review what any exposure is.

Appearing to be upfront and cooperative, which as I recommend one should do for many reasons, dispells the myth that you stole something of value. This is the presumed hope of anyone wanting to catch a company offside; this is the whole business plan of a patent troll. Watch someone build something of value, count up the toll, and then strike with a lawsuit to demand payment.

A consistent, well documented, and clear software rationale, well documented software source code, an updated library of the improved / augmented code, released at regular intervals into the public domain. This becomes a justification in and of itself. In fact, the ability to do this for the greater software good is no different than dealing with many other issues like security, estimating code value, and so on.

Remember, I am the veteran of reviewing and attempting to reuse over 5000 software projects, I have invested many hours understanding and then reflecting on the problems of software code, so the view in this book isn't a hypothetical from a neophyte graduate.

If I was asked to counsel a judge about when a source file has changed from the original copyright / legal terms and conditions may no longer apply, it's when the code and comments inside a single file exceed 51% of the original in content and form. Of course, the simplest way to avoid file copyright infringement is to place all new code in a separate file, but there are valid technical reasons why people would want to keep the code in a file scope, and fair use doesn't automatically prevent the ways you can use a work. That is the grey area a judge would be privy to consider. The reason I call it 51% is that while the ideas inside may have been understood enough to copy the code without the code and outside, is that the original content, whatever it was, at some point diminishes to the point it's like a public function that people explain to each other online; it ceases to be an original idea. Software code, by the rules of syntax and grammar, is mainly pro forma language. The names of things inside the code, object names and function names primarily, are the majority of independent ideas painted on the canvas of code. At some point, you have painted over the canvas with a majority of new ideas that you can't recognize the original. And that's when some common sense and fair use should prevail if we want society to get the most benefit of salvage in code as in ship salvage.

Now, in all likelihood you will encounter no one desiring to make a federal case about a grey area of fair use. Is there really anything to a complaint that something of value that was improperly used? The biggest justification for your work is to demonstrate just how much better you made the code; making concrete improvements gives this a massive quantum leap in the re-use of software for generations to come.

Of course, the one kind of software source code or libraries, that must be used with caution is the commercial software, commercial software has a developer team and software adherents inside and outside the company. This is the one kind of software you should probably just use separate and apart from anything else. Now, that being said, there are many cooperative uses of commecial software that appear when that company abandons a specific code base. I can think of my use of protobuf-c in the wild. Google once began and ran the software called protocol buffers that was developed by Google first in C. Once they decided to end support for the code, and they announced and left it within the archives for a few years. Then, they stopped offering the C version. It disappeared from the Google archives. Instead, they allowed only the Java, and C++ and other versions. I have worked with the protobuf-c open source forked and placed on the git repository website for many years. I use protocol buffers as my marshalling and demarshalling software libraries in my robotics work; I got fed up with the code bloat of C++ and MIRO in particular. Daily, I keep a listening watch on that code base and get the bugs and error into my email feed. Not once has any work been stopped or waylaid by Google. As far as I can tell they never went after the git operators either. In this small case, Google haven't been evil. So there are many kinds of end of life cycle source you can lay claim to when the time is right and the need is real. Lots of large software houses purchase companies and realign the work between two sides of the merger - and inevitable layoffs. There will be code you can obtain free in this way. Who knows, you may find people willing to fix their old code out of allegiance to their work?

So while the legal terms may be daunting, all you need to do is simple: find a dirty penny in the grass and shine it up. You attempt to contact them about that shiny penny and be willing to share it back with lost owners. The worst thing you can do if you have appropriated code that may be salvage is ACT like you are stealing something.

The one unifying act that you can do to preserve a positive perspective, is to leave a software library of the unclassified areas of your work, into the public domain. Given the idea of value as a justification for software re-use, as long as you align with the long term goals of software advancement, is to offer that better code in limited libraries to other parties so that they can stand on your shoulders, to perpetuate the acceleration of software reuse.

If and when you get ready to commercialize, you should do a code review to determine all the code that originated within these separate properties. You identify the code as new or derived from existing code bases so you are prepared for any dispute by the original coders' complain.

In all, take this recommendation with a grain of salt. I will harken back a few years when Samsung and Apple were engaged in open legal warfare over violations of patents on both sides. There they were, Apple and Samsung, both violating property and escalating their ultamtums and threats with each passing lawsuit launched at each other. The sums in question were sizeable, and the word got out on Wall Street. Both companies were facing liabilities and tarnish to their reputations and brands. And then, as violently as it begun, there was silence. Each side, realizing the downward spiral it all made in toll, as other competitors looked on, was enough to make them see green. It always seems a good idea to charge out into the legal battle, until the cooler heads in accounting start warning of the impending carnage. It's hard to show a profit per unit with all those lawsuit holdbacks from profits. It's harder to make your stock options.

Take all of this with a grain of salt, and if you are uncertain consult a lawyer with intellectual property expertise and experience to be certain.

Chapter 16

Re-Use Library Abstraction (RULA)

The gitlab library RULA can be found here:

RULA

This project houses the (to be confirmed) IEEE Standards Association Proposed PAR (Project Authorization Request) entitled: Re-Use Library Abstraction (RULA) as a means to accelerate reuse library adoption by decreasing library learning uptake time. The goal of this work is to implement a Basic Prototype Exemplar that holds all the factors recommended in the book: (TBP - to be promulgated) The goal is to showcase the easier to use, easier to understand, and easier to improve aspects that will make it win over users through time-tested improvements to the way things are done to this point.

This section decries the micro level details of the software and associated files within the RULA system. That's why this library is different: it has the foreground process you are intimately familiar with on storing code for compilation, but it also has a background process that does work on files in the background when they are inserted to give an overall status to those that access it.

Chapter 17

Conclusion - What's at stake?

Let's circle back (a very cliched and currently expected phrase - linked mostly to virtue signaling and consequent failure) to the heart of the issue. What was the goal of this book?

> This view of philosophy appears to result, partly from a wrong conception of the ends of life, partly from a wrong conception of the kind of goods which philosophy strives to achieve. Physical science, through the medium of inventions, is useful to innumerable people who are wholly ignorant of it; thus the study of physical science is to be recommended, not only, or primarily, because of the effect on the student, but rather because of the effect on mankind in general. Thus utility does not belong to philosophy. If the study of philosophy has any value at all for others than students of philosophy, it must be only indirectly, through its effects upon the lives of those who study it. It is in these effects, therefore, if anywhere, that the value of philosophy must be primarily sought.
>
> Professor Bertrand Russell[1]

Software reuse is like philosophy. It doesn't seem to make any sense or have any purpose for most but if it isn't in the background improving software for their technology - most people's interface with science - then the impact is felt in less safe products and more costly units. The impact is there whether you realize it or not.

The information in this book makes for a better start to your work; an attempt to give you more sleep, more observers transformed into adherents to your shared efforts, more ease of use through better defined knowledge explained in

[1] https://www.gutenberg.org/ebooks/5827

more than the first blush surface meanings to reduce confusion, a better search function for others to find what they need, or confirm you don't have what they seek (but fill them with an impressed and positive experience), some wise guidelines about the other dimensions that will impact your survival at the code level over the longer term, and your ability to not start from zero.

Starting from zero is the start state we work to avoid for everyone starting off in software in the future. We want to cure humanity of this perilous wastage of time. The stakes are implanting a major, thoughtful, new way to evolve software code by leveraging my failed experiences, my revelations on how to avoid these pitfalls in the future, and for everyone in mankind to benefit from your work by making it relevant into the future.

This book, like any receptacle of knowledge, is a best attempt at arming you with the current, direct, and timely information that can assist you.

I will make myself available for questions, comments, queries at Veterans-DefenceInvestment@gmail.com. Please send me a line if that's the only push needed to send you over the tipping point.

The rest is up to you.

Let me leave you with the very prescient words that Professor Douglas Schmidt about the struggle towards common software and reuse:

> *"I have repeatedly witnessed organizations that initiate systematic reuse efforts with the best of intentions, only to lose faith when various impediments arise or schedules slip. Inevitably, they then fall back onto familiar processes, i.e., developing their software from scratch. I've observed that reuse-in-the-large is best achieved when development and management leaders are unwavering and evangelistic. Moreover, this faith must be propagated up to, and echoed by, the highest levels of the organization.*
>
> *Ultimately, organizations that attempt systematic reuse without providing an incubation environment will lose their faithful. Many of these faithful will be the most experienced developers or those most capable of coming up to speed quickly. In markets driven by "Internet cycle times," the loss of valuable developers can devastate an organization's long-term competitive viability.*
>
> *Keeping the faith requires keeping abreast of external R&D developments and global technology trends. In my travels throughout the software industry, I am continually amazed by the rate at which reuse and COTS middleware is being adopted in many businesses and application domains. I suspect that the pundits who dismiss reuse as a myth simply haven't spent enough time "in the trenches" lately to recognize the speed at which the software industry is moving away from programming applications from scratch to integrating*

applications out of reusable frameworks and components."

Professor Douglas Schmidt, in 1999.[2]

[2] **Keep the Faith: https://www.dre.vanderbilt.edu/∼schmidt/reuse-lessons.html**

Appendix A

Acknowledgments

Thanks to my family, far flung but still connected by blood and love. You stay in my thoughts long enough to remember what makes my life special, and well worth it.

To Darcy, my bright young hard working daughter.

To Rayleigh, my worldly and wise daughter.

Appendix B

All-Encompassing Index

In order to contain knowledge in the preceding work in one kind of index, and to cross reference things on many levels, this **All-Encompassing Index** aims to make finding that idea on the tip of your cognition faster and easier. The aim is better association and thereby better recognition. It is hoped that by your ability to reference ideas faster this way, you will agree that the association of ideas like keywords directly linked to code will make better software a result.

For example, say you were trying to remember the fair use considerations presented in this book, and you recall that the one you think you want (which was known to you by keywords you understand) is nearing to the part with Mickey Mouse in it than the part about pictograms. This is all perfectly normal attempted association of one known (you remember) idea with another idea that you are certain is close by. Then you look in the all-encompassing index to find the one Mickey Mouse reference (because I only included it once in the text of this book) and it brings you to within striking distance of where you want. I tried to link most key concepts in this book with external, non-software concepts or ideas, and in this way expand the memories available to your success. Just like I repeated the same idea verbosely here to make it plain in many ways.

That mental mechanism you are exploiting is the key rationale behind concept-idea-keyword association for finding information readily in the software library as well.

Nomenclature

infographic	A visual representation of information..
LOC	Lines of Code
pictogram	A picture that represents a word or an idea by illustration.

Index

Bibliography

[1] ISO/IEC 24707. Information technology- common logic (cl): A framework for a family of logic-based languages, 2007.

[2] John W Bailey and Victor R Basili. The software-cycle models for re-engineering and reuse. In *Proceedings of the conference on TRI-Ada'91: today's accomplishments; tomorrow's expectations*, pages 267–281, 1991.

[3] Victor R Basili. Maintenance= reuse-oriented software development. In *Conference on Software Maintenance*, number UMIACS-TR-89-48, 1989.

[4] Victor R Basili and H Dieter Rombach. Towards a comprehensive framework for reuse: A reuse-enabling software evolution environment. In *NASA, Goddard Space Flight Center, Proceedings of the Thirteenth Annual Software Engineering Workshop*, number UMIACS-TR-88-92, 1988.

[5] Richard P Feynman, Michael A Gottlieb, and Ralph Leighton. *Feynman's tips on physics: reflections, advice, insights, practice*. Basic Books, 2013.

[6] Bill Gallmeister. *POSIX. 4 Programmers Guide: Programming for the real world*. " O'Reilly Media, Inc.", 1995.

[7] Pete Goodliffe. *Becoming a Better Programmer: A Handbook for People Who Care About Code*. " O'Reilly Media, Inc.", 2014.

[8] Brian J Gough and Richard Stallman. *An Introduction to GCC*. Citeseer, 2004.

[9] ISO/IEC. Programming Languages - C - Extensions to support embedded processors. Technical Report 18037, International Standards Organization/ International Electronics Communication, July 2004.

[10] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[11] J Marshall, Stephen Berrick, Angelo Bertolli, Corey Bettenhausen, Howard Burrows, Saurabh Channan, Victor Delnore, Robert R Downs, Yonsook Enloe, Stefan Falke, et al. Reuse readiness levels (rrls). *sciencedatasystems.org*, 2010.

[12] Peter Seebach. Beginning portable shell scripting. *New York*, 2008.

[13] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[14] Richard M Stallman et al. Using the gnu compiler collection. *Free Software Foundation*, 4(02), 2003.

[15] Will Tracz. Where does reuse start? *ACM SIGSOFT Software Engineering Notes*, 15(2):42–46, 1990.

www.editorium.com
editor@editorium.com