COMPLEXITY & VERIFICATION:
THE HISTORY OF PROGRAMMING AS PROBLEM SOLVING

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Joline Zepcevski

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arthur L. Norberg

February 2012

Acknowledgments

It takes the work of so many people to help a student finish a dissertation. I wish to thank Professor Arthur L. Norberg for postponing his retirement to be my advisor and my friend over the course of this project.

Thank you to my committee, Professor Jennifer Alexander, Professor Susan Jones, Dr. Jeffery Yost, and Professor Michel Janssen, all of whom individually guided this dissertation at different times and in specific ways. Thank you also to Professor Thomas Misa for his guidance and assistance over many years.

I had a great faculty and a great cohort of graduate students without whom this dissertation would never have been completed. I particularly want to thank Sara Cammeresi, whose unending support and friendship were invaluable to the completion of this project.

I wish to thank my family, Jovan Zepcevski, Geraldine French, Nicole Zepcevski, and Brian Poff, who supported me and loved me throughout it all. I also want to thank my friends: Tara Jenson, Holly and Aaron Adkins, Liz Brophey, Jennifer Nunnelee, Jen Parkos, Vonny and Justin Kleinman, Zsuzsi Bork, AJ Letournou, Jamie Stallman, Pete Daniels, and Megan Longo who kept me sane.

I need to thank Lisa Needham for all her assistance. Without your help, I wouldn't sound nearly as smart. You're a great friend and a great editor! Thank you also to Karl Brophey for his feedback.

I was lucky to have a great early experience in this field at the University of Sydney, under the leadership of Rachel Ankeny and, later, at the University of New South Wales with David Miller and John Schuster.

Finally, without the early guidance of a dedicated and caring professor, Dr. Katherine Neal, I would never have pursued this path. I was so lucky to have had a teacher who could make such an enormous impact on a student.

For my mother, who taught me everything I know – including how to use a computer.

And my father, who gave me the opportunities he never had.

Complexity & Verification:
The History of Programming as Problem Solving

Abstract

Changes in computer programming methods were responses to specific stimuli, and that (contrary to much existing analyses) the development of programming methods does not fit an ideal of "progress." I focus on the rise of two fundamental computing problems: complexity, or the proliferation of people and methods; and verification, which is the (in)ability to verify that a program functions as intended. Complexity and verification were the catalyst for the development of automatic coding systems but also increased exponentially as a result of automatic coding systems like FORTRAN and COBOL. These systems have English-like commands that simplify programming. The adoption of automatic coding systems opened up the programming field to more software engineers and allowed the creation of more elaborate software systems, creating ever more complexity in the discipline. I argue that since the introduction of automatic coding systems in the 1950s, methodological changes and new programming languages have been attempts to solve long standing problems faced by programmers. Not, as the traditional insider narrative suggests, a steady evolution based on a better understanding of programming. In this dissertation, I focus on the changes motivated by two stimuli — complexity and verification.

Contents

Table of Figures

Chapter 1 Overview: A Case Study of Complex Technological Change

Introduction

Modern society has become dependent on computers and the software that drives them. Transportation, from traffic lights to air traffic control; health care, from digital thermometers in the home to sophisticated medical machinery in an intensive care unit; communications; financial and manufacturing systems – the systems that modern society depends upon are all driven by software. Yet, almost from the moment of inception, before it had subsumed any of these systems, there were claims that the computer was revolutionary.

Claims of a computer revolution, common in the early history of computing, have been transformed into the current day "Information Revolution" rhetoric we see in popular culture. This transformation is indicative of a changing understanding of computing as a whole. In the early years of computing, hardware was seen as the primary obstacle to the use of computers; with time it became obvious that software was equally as important. Without software to drive the computer, the computer was, at its essence, only a calculating machine. Software allows the computer to be a multi-purpose

machine by collecting, collating, organizing and processing information into useful formats.

Managing information has been a part of the human search for knowledge since antiquity. However, as Thomas Haigh demonstrates in his dissertation, "Technology, Information and Power: Managerial Technicians in Corporate America, 1917 – 2000," our modern definition and understanding of information has developed alongside the digital computer. Information is a reinterpretation of the verb "to inform." Information science, a term for specialized library work and information theory, first conceived by Claude Shannon, flourished. Information, as a collection of facts, became a part of business organizations as the computer took on an ever-increasing role.[1]

Information has become a fundamental scientific and technological concept. These new ways of using information have reshaped disciplines and created new fields of inquiry.[2] The change in rhetoric from the computer revolution to the information revolution illustrates our changing conception of the importance of the role of software. Software is no longer the simple translation of human commands into a language the computer understands. Instead,

---

[1] Thomas David Haigh, "Technology, information and power: Managerial technicians in corporate America, 1917--2000" (Ph. Diss. University of Pennsylvania 2003) 400-406

[2] Michael S. Mahoney, "The History of Computing in the History of Technology." *Annals in the History of Computing* 10, no. 2 (1988): 113-125.

software and hardware are interdependent parts of a computer system. The development of both hardware and software deserve attention as technologies that have deep relationships with our society and culture. My dissertation focuses on the development of software.

What I find most interesting about software (which is presented as a part of the computing monolith that revolutionized society and has subsumed so many of society's systems) is that it is often claimed to be undergoing a revolution itself — the way software is written is being revolutionized. That claim is the focus of this dissertation. Throughout the brief sixty-three year history of the software field there have been repeated claims of revolution within the discipline. Yet, many of these revolutionary claims rested on future promises, not existing performance. This dissertation will address these claims by asking if the way programmers write software has changed in a dramatic fashion between the 1950s and the 1980s. It will explore why changes occur, what those changes look like, and whether they are revolutionary or evolutionary changes.

Traditionally, like the computing community as a whole, the community that creates software has presented their own history as a technologically optimistic narrative. This insider-driven narrative portrays the field as progressing towards an epitome of software

design, methodology, and language. Their constant claims of revolution in the literature are then reinterpreted later as evolutionary changes — small steps on a road to the best and only true way to write software. This deterministic narrative can be inferred from trade publications, like *Datamation* and *Byte*. It is also seen in the *ACM Proceedings of the History of Programming Languages* conferences. Historians of computing re-evaluate these deterministic narratives, using these sources as primary literature. With this rich primary literature, historians can then present more subtle and multi-faceted stories of change. Like other works emerging from the history of computing, this dissertation will re-evaluate this insider-driven narrative and present an alternative story of the history of changes in software design, methodology and language.

Why is this story important? Why should you find this story interesting? It is important because a history of the development of software illustrates that the nuts and bolts of the way software is written is not following a path but, like many other histories of technology, it is instead created by a number of different interactions between people, the machines, and the prevailing fashions of the field; influenced by and influencing a larger socio-economic milieu. This should be interesting to the historian of technology because it

demonstrates symmetry with the development of traditional (physical) technologies.

Within the history of technology, as technology has become more complicated, the lone inventor paradigm has essentially disappeared. More modern technologies are created through the collaborative workspace and incremental invention of commercial laboratories. This dissertation illustrates that the creation of software is still a space where we can see overlap between the lone inventor, the inventor's workshop, and the group work of a commercial laboratory. It has progressed through what I call a collective tinkering, with feedback loops that provide guidance for the direction of the field. Programming theory can still be swept towards a new methodology or programming paradigm by a charismatic leader. The field is young and dynamic enough that lessons from the past are sometimes integrated (with new names and explanations) into the newest development, while at other times the methods of the past are washed away without a glance, with little explanation for why they fell out of favor.

However, in the primary literature, when retelling their stories, individuals often justify themselves using stories from history. At times they adopt the cloak of scientific method and at other times use

a modernized lone inventor or "ingenuity" tale to justify their positions. Programming specialists use metaphors, comparing the field to the American system of mass production or the assembly line, and expecting that if they can just impose order on the chaotic programming field they could manufacture code just like Ford or Singer. At times, this story, stripped of its sense of progress, can seem terrifying —like the idea of a power company creating a network without a grid that still somehow provides electricity, with all the possibilities for dramatic failures that accompany the history of electricity.[3] At other times, the story feels like a pioneer story complete with heroes and villains. What the history of software never feels like is a story of technological progress, moving inexorably towards a predefined set of goals or outcomes.

This dissertation is a case study of complex, technological change — people working on the edges of their disciplinary boundaries and an industry undergoing growing pains. I focus on the technical changes in the methods, designs, and languages programming specialists use to write software: what those changes look like, why they happened, and their ramifications. This argument parallels many of the histories of software, focusing on a similar chronological period.

---

[3]Donald MacKenzie, "Computer Related Accidental Death," in *Knowing Machines* (Cambridge: MIT Press, 1998), 185-214.

As a result, I see my work as complementary. I focus on the technical changes that were happening in parallel to the debates about professionalization, social hierarchy within the broader software community, the software industry, and the spread of software to new fields, so well covered by the existing historical literature.

It is often repeated that historians of computing know much about the history of programming languages, but little about other aspects of software.[4] In this dissertation, I refute this argument – we know a lot about the facts, figures, and dates of how new programming languages formed within a bubble. Much of this is primary literature, written by participants. However, I argue that we know very little about the way programming languages formed in the broader context of the field. Why do we have so many programming languages? What are the real benefits of one programming language over another? Why would one language become an overnight success, while another language languishes with a small following? What prompts a programmer to begin creating a new language? What

[4] Michael Mahoney, "Issues in the History of Computing," in *History of Computing: Software Issues*, ed. Ulf Hashagen, Reinhard Keil-Slawik, Arthur L. Norberg (Berlin: Springer, 2002), 1-5. I further argue that this has become dogma since the 2002 Paderborn Conference, without further considering the subsequent statement: That there is little appreciation for the overall history of software. The purpose of this dissertation was to produce a broad, overarching story of changes in programming, a goal I believe fits into the research agenda produced at the Paderborn conference, particularly the guideline that we should place development in a larger context and that historian[s] should not see change as an inevitability of the software field, instead it is a part of broader changes– the philosophy, the institutions, the educational fashions, economic factors and professionalization.

influences their decisions when they begin creating the syntax?[5] These are the questions I will address in this dissertation.

There has also been a push within the history of computing to address software applications. While I believe that a history of software applications will add to the literature, I argue that a sophisticated analysis of programming languages will add to our understanding of applications. Applications are written in programming languages. These languages define how the application can be imagined, the world view in which the programming specialist works. When creating software, the programming language is the tool, while the application is the result. To draw on the manufacturing metaphors used by computing professionals in the primary literature, the programming language is the system of mass production: the lathe, the jig, the gauge; while the finished application (piece of software) is the finished product – the gun or the sewing machine. Both are good topics for investigation, but just because we have a significant amount of primary literature on programming languages does not mean that we should not be looking for more nuanced

[5] These are questions that have arisen in the literature; Nathan Ensmenger has an entire chapter devoted to the question of why are so many programming languages. I argue that there are multi-faceted answers to these questions, which include arguments about the social and economic needs, and also assess the underlying theoretical schisms that make one programming language suitable for a project, but another language unsuitable.

explanations and explorations of change in the tools programming literature use to create software applications.

My research has illustrated that there are many technical stimuli that drive technical changes in software design, methods, and languages. Efficiency, a concept with its own rich and complex history, portability (the range of hardware and operating system platforms on which the application can be executed), usability (the ease that a person can use the program), and maintainability (the ease with which a program can be modified) can all be seen in the literature as stimuli for change. In this dissertation, I am focusing on two stimuli, complexity and verification, which reappear throughout the history of software, from the inception of the discipline through today. Complexity and verification are not merely catalysts. By tracing the literature and comparing it with the changes made in the source code itself, complexity and verification have an explanatory power for describing the specific technical changes that occurred in program design and implementation.

Definitions

One of my challenges when writing this dissertation was to clearly define several terms that are at the core of my argument. Four terms were particularly difficult to clearly define. Most difficult is

defining the community of software practitioners that are at the center of this narrative. All of the practitioners in this story describe themselves as "computer scientists" or "programmers." Often they used these terms interchangeably. After the introduction of the term "software engineering" in the late 1960s, members of the software community adopted the software engineer label as facilely as they had adopted earlier terms and used them equally interchangeably. The difficulty with these terms for my needs is that they encompass many different parts of the software field – graphics, networking, database programming, and software metrics are all subsumed under the computer science and software engineering headings. While some of these sub-disciplines have relevancy to this dissertation, others are completely separate. As a result, I will redefine the community I am focusing on as **programming specialists**. Moreover, there is a question of what the community that I am writing about is working on - programming languages, programming tools, programming methodology. The work ranges across these different areas and so to give myself a broad platform I am defining this as **programming theory**. Even the concepts at the very heart of my argument, verification and complexity, while seemingly straightforward, are conceptually difficult terms. Further, both verification and complexity have different definitions in different contexts and at different times

throughout the history of their use in the field. For the purposes of this dissertation I am defining **complexity** to be the proliferation of people and an increase in the scope, scale, and algorithmic complexity. I am defining **verification** to be the (in)ability to verify that a program functions as intended. To resolve the conceptual difficulty of defining these terms, I will discuss them in relation to the existing literature and clearly define how I will use these terms throughout the dissertation.

Complexity

It is difficult to define complexity in the abstract. In every day usage, society asserts that something is complex if it is made up of many intricate and interconnected parts.[6] However, for the history of software, this is an empty definition. What are these intricate and interconnected parts? What do these parts add up to? When turning to *A Dictionary of Computing* to find a more specific definition we see that complexity is defined in terms of computational complexity – the difficulty of solving a computational problem, measured by the resources consumed during the computation.[7] Yet this very specific definition is too narrow for the argument that I am making. As I have

[6] "complexity" Merriam-Webster.com. 2011. http://www.merriam-webster.com (June 1, 2011).

[7] I.C. Pyle and Valerie Illingworth, eds., *A Dictionary of Computing, 4th Ed*(New York: Oxford University Press, 1997)

illustrated throughout this dissertation, the actors focus on different aspects of complexity and using a broader definition will allow me to capture a more complete picture of complexity as a stimulus for change. When examining the literature in the history of software, it became clear that while authors like Ensmenger, MacKenzie, and Shapiro all discuss the complexity of software, as it is not their core argument they don't offer a definition for what makes software complex. This is similar to the problem in the primary literature. While there is much talk about the complexity of software it is always in terms of a problem to be surmounted. For example, William Orchard-Hayes' article that is devoted to "solving the programming problem", in which he defines the problem to be (in part) complexity of programming and the increased complexity of applications desired, he never defines what complexity is – he instead moves on to define how we should resolve complexity – through improved communication and management. I redefine complexity as a stimulus for change in programming and to do so, I need a specific definition of what complexity means. Here, I define complexity: In software, the interaction among the people working on the project, the scope and scale of the program, and the algorithmic complexity combine to create the overall complexity of a software project.

People

Software is human activity and, while it can be seen as an indirect variable, there is evidence demonstrating that people can add complexity to a software project. The IBM System 360 project is a well-known example of the way people can increase the complexity of a project. Nathan Ensmenger illustrates in his work that complexity arises in the interaction between the person and the machine. He cites Brooks when arguing that the physical act of translating the written or spoken word into symbols the machine can understand is a difficult process. Computers demand perfection and precision in a way that few other media require.[8] Brooks illustrates the difficulty that arises as more people become involved in a project. The management of a larger number of people requires increasingly formal methods of communication between people, particularly when they are geographically separated.[9] Looked at more broadly, the history of science illustrates the complexity of transferring tacit knowledge. For example, H.M. Collins illustrated the difficulty of transferring tacit knowledge in the construction of TEA lasers in geographically disparate

[8] Nathan Ensmenger, *The Computer Boys Take Over*, (Cambridge: MIT Press, 2010), 45.

[9] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Reading: Addison-Wesley, 1975) 14 – 26, 61-81.

areas.[10] This is equally visible in the software literature. In 2002, IEEE published an article aimed at solving the problem of tacit knowledge, "Replicating software engineering experiments: addressing the tacit knowledge problem," which illustrated these problems of communication and tacit knowledge in software design.[11] Furthermore, the Standish Group's well-known 1995 Chaos Report argued that the second highest cause of failure in software projects was incomplete requirements – essentially the inability to clearly communicate the clients' goals to the programmers.[12] In all of these ways, the role of people in the creation of software can increase the complexity of the project.

Scope

For my purposes, scope is the breadth of the processes the program is designed to control. There are two aspects of increasing scope. The first is the naturally occurring increase in the scope of successive projects as hardware improves and we, as a society, increase the use of computers in our lives. The second aspect is when

---

[10] H.M. Collins, "The TEA Set: Tacit Knowledge and Scientific Networks." *Science Studies*. 4. no. 2 (1974): 165-185.

[11] F. Shull, V. Basili, J. Carver, J.C. Maldonado, G.H. Travassos,M. Mendonca, S. Fabbri, "Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem," in *Proceedings of the 2002 International Symposium on Empirical Software Engineering* (Washington, D.C.: IEEE Computer Society, 2002): 7-16

[12] Standish Group, *Chaos Report*, 1995 (http://www.projectsmart.co.uk/docs/chaos-report.pdf)

the scope of an individual project "creeps." For example, when management requests alterations to a project that has already been defined, these changes increase the breadth of the project. Brooks describes this in *The Mythical Man Month* as putting "Ten Pounds in a Five Pound Sack."[13] More recently, *IEEE Spectrum* completed a case study on the FBI Virtual Case File failure, where changing requirements ("scope creep") were a significant part of the project's failure.[14]

Scale

When talking about scale, I quite literally mean the number of lines of code in the project. As IEEE reported in *Spectrum* in 2005, "A project's sheer size is a fountainhead of failure. Studies indicate that large-scale projects fail three to five times more often than small ones."[15] As both applications and systems software become broader in scope, there is a corresponding increase in the amount of code. As computers become faster and software applications and systems software are expected to do more things automatically, there is a corresponding increase in the amount of code. The more lines of code,

---

[13] Brooks, *The Mythical Man-Month* 98 - 103

[14] Harry Goldstein, "Who Killed the Virtual Case File? How the FBI blew more than $100 million on case-management software it will never use." *IEEE Spectrum* (2005)

[15] Robert N. Charette, "Why Software Fails. We waste billions of dollars each year on entirely preventable mistakes." *IEEE Spectrum*, September 2005

the more interactions between different parts of the program, and the more decisions the software makes, combine to increase the complexity of the program.

Algorithmic Complexity

Algorithms are the rules or instructions for solving a problem in a finite number of steps. This set of instructions can be mathematically complex in that it can include many steps in the program, but also because programs are inherently dynamic. Programs branch, they loop - all of these dynamic elements increase the complexity of the algorithm. Moreover, in most software systems, there are many algorithms. For example, a sorting algorithm may sort a list to find one piece of data. That is a single algorithm, but if you then want to transform that piece of data using a looping structure, for example, that loop is a second algorithm. Moreover, there is always, of course, any number of ways to approach a problem and achieve the same ends. So, different sorting algorithms, different branching algorithms, and different loops all have different levels of efficiency and complexity. These algorithmic metrics influence the performance of the final program.[16]

---

[16] The complexity of algorithms has spawned several theoretical sub-disciplines in computer science. Throughout my dissertation I will talk about formal mathematical verification of programs (and algorithms). There are three other sub-disciplines that revolve around

The complexity of software is governed by the interaction of these elements: people, scope, scale and the algorithmic complexity of the project. The interactions between these interconnected parts are difficult to predict. Changing one element can change the others. Consequently, the complexity of software is always a moving target.

Verification

Similarly, the definition of verification is also a moving target. Using *The Oxford Dictionary of Computing*, in the 3rd, 4th and 6th editions, verification is defined as the process of checking the accuracy of the transcription of information. *The Oxford Dictionary* then specifically applies this definition to data verification. Data is the information the program works on, for example, data that would be entered into a database or spreadsheet system. Yet, when reading the primary literature in the history of software, it is clear that verification, to those working in the field, has a far broader meaning than merely that of data verification. The dictionary definition that most relates to the language in the literature is the concept of "verification and validation." This is defined as the complete range of checks used to increase confidence that the system is suitable for its intended

---

algorithms: (1)Algorithm computability – illustrating that algorithms are computable; (2) Calculating the efficiency of an algorithm (how long it will take the algorithm to perform the computation); (3) Calculating the complexity of an algorithm and then ordering those algorithms in classes of complexity. Together, these can be considered algorithmic metrics.

purpose. These two definitions both differ dramatically from the functional definition used in the early years of the software discipline, when verification was linked inextricably to formal mathematical verification techniques. However, the broad definition of "verification and validation" illustrates one of the most important conclusions of this dissertation – that as a result of the ever-increasing complexity of programs, the definition of verification has itself changed to encompass a wide range of techniques, not only formal mathematical verification techniques.

I want to use a broader definition from that in the *Dictionary of Computing* and, with that in mind, I am also drawing on MacKenzie's description in *Knowing Computers*. MacKenzie uses the term "formal verification" to refer to the formal mathematical verification techniques promoted in the early software literature. MacKenzie defines "formal verification" as the appropriation of the concept of mathematical proof. A mathematical proof is an argument that demonstrates that a theorem holds in all possible cases. In computer science, formal verification became the process of creating a mathematical representation - first a representation of what a program or hardware design was intended to do and then creating a mathematical representation of what the program does do. You could then verify that the program accurately did what it was supposed to do using

mathematical deduction.[17] This is often, but not always, the working definition used in the software literature throughout the late 1980s.

For this dissertation, it is important to make a distinction between verification and testing. The *Dictionary of Computing* defines testing to be any activity that checks whether a program, system or component behaves in the desired manner, by means of actual execution. The key to this definition is "by means of actual execution," which differentiates testing from formal verification. Testing is conducted by test runs, where the system is supplied with input data and the responses of the system are recorded for analysis.[18] There are many ways of organizing the testing process.[19]

I make this distinction because there was a division within the software community about the best way to verify programs – namely, whether that should be formal verification techniques or testing techniques. The pendulum swings between these two perspectives for much of the time period examined in this dissertation; moreover, by the late 1980s the definition of verification has itself changed to encompass both formal verification and testing techniques. For clarity,

---

[17] MacKenzie, *Knowing Machines,* 4 -7

[18] I.C. Pyle and Valerie Illingworth, eds., *A Dictionary of Computing, 4th Ed* (New York: Oxford University Press, 1997)

[19] Different techniques include black boxed testing versus component testing, branch testing versus path testing and others.

I will use the term "verification" to refer to the broader practice of verifying that a program is accurate to its specifications, including both using mathematical proofs to deduce the accuracy of and practical testing techniques. The term "formal verification" will then refer explicitly to mathematical verification techniques. "Testing" then refers explicitly to testing techniques.

Unlike complexity, the role of verification has been defined in the history of software. Arthur Norberg illustrates the role that the artificial intelligence community, a sub-branch of computer science, played in bridging this gap between mathematical proofs and testing techniques. The AI community focused on ways to automatically prove the accuracy of programs. An entire series of "boutique" programming languages were developed by the AI community, with an eye to developing a language capable of being subjected to automatic, mechanical proofs.[20]

In Donald MacKenzie's *Mechanizing Proof*, he describes how verification is a socially constructed concept. This dissertation uses that definition, and further illustrates how the understanding of verification has been renegotiated over time, from the very specific formal mathematical meaning of the 1960s, through to the more

[20] Arthur Norberg and Judy O'Neil, Transforming Computer Technology: Information Processing for the Pentagon, (Baltimore: Johns Hopkins University Press, 1996), 197 – 213

liberal meaning of verification techniques and testing used in recent work. In "Fangs of the VIPER", chapter 7 in *Knowing Machines*, MacKenzie recounts that in 1986, the UK's Cabinet Office Advisory Council for Applied Research advised that mathematical proof should be mandatory for any system that, if it were to fail, could result in more than ten deaths. The UK's Ministry of Defense created the VIPER (Verifiable Integrated Processor for Enhanced Reliability) chip, believing that it met this requirement, because there was a mathematical proof of the correctness of the VIPER's design. However, contention over what could be considered a proof drove the matter to court. Unfortunately, the courts did not get a chance to hear the case, because one of the litigants went bankrupt, but the tale does demonstrate that mathematical verification is not as cut and dried as it sounds. One person's proof is another person's rhetoric.[21]

In chapter 9 of *Knowing Machines*, "Negotiating Arithmetic, Constructing Proof", MacKenzie followed an earlier debate in computer science. In 1979 Richard De Millo, Richard Lipton and Alan Perlis spoke about the impossibility of using automated mathematical proofs to prove the correctness of hardware or software.[22] This led to

---

[21] MacKenzie, "Fangs of the VIPER," in *Knowing Machines*, 158-164.

[22] MacKenzie, "Negotiating Arithmetic, Constructing Proof," in *Knowing Machines*, 178-183.

significant debate in the programming community, one that I trace avidly in the fourth chapter of this dissertation.

These stories suggest questions for historians of technology to pursue. For example, what does a proof prove? In software, mathematical proofs can only prove that the design of the software is correct, not that it was correctly implemented. This, again, leads to the question: what is proof? My dissertation takes these questions as a starting point and expands upon them, exploring the role of verification as a stimulus for change in programming designs, methodologies and languages.

Defining the community

In my narrative, I discuss a particular community of practitioners, but carefully defining this community is difficult. The existing literature in the history of software is only preliminarily helpful in defining my community. Haigh's dissertation delineates a number of sub-categories in the software community, describing their roots in corporate America and their evolution to different areas of expertise.[23] However, all of these sub-categories can be described using Ensmenger's term: vocational programmers. Ensmenger's work

[23] Haigh, "*Technology, information and power*," 1 – 4.

focuses on the tension between these vocational programmers and what he defines as academic computer scientists.[24] I am focusing on these academic computer scientists, or as I prefer theoretical computer scientists, as these actors work in both private and academic institutions at different times in their careers. These actors may take jobs in the private sector, such as Harlan Mills or R.V. Head, but at their root exhibit all the characteristics of a theorist - holding graduate degrees, contributing to the literature, and attending conferences, reflecting the behavior of any other theoretical field. Tensions clearly exist with vocational programmers and computer users, as evidenced in both Ensmenger and Haigh's work, but this dissertation focuses on tensions among theoretical computer scientists.

Within this broader group, I must delineate a smaller group that I will call programming specialists. These programming specialists enter and leave the narrative, moving freely among a number of different sub-specialties (operating systems or applications, for example) over the course of their careers. Throughout this dissertation I focus on the tensions that occur between the programming specialists. This community has a fluid dynamic; creating different alliances depending on the debate, but throughout

---

[24] Ensmenger, *The Computer Boys Take Over*, 11 – 14.

the period of this dissertation the most overarching tension is between those specialists who are pursuing formal verification as the solution to problems in software and those pursuing testing and practical techniques to solve these problems.

Defining the topic

Finally, these programming specialists work on ranges of topics related to the theory of programming.[25] The work is not limited to defining and implementing a programming language. The actors work on programming methodologies, programming tools, and programming languages. Programming languages are the systems of notation that programs are written in and their implementation into machine language (through an interpreter or compiler). Programming tools are a broad topic in and of themselves; they are technologies of support that ease the burden of programming. These can be debugging programs or maintenance programs. They can include verification tools. Programming tools can be combined into sophisticated software packages. However, tools can also be as simple as paper flowcharting techniques. Finally, programming methodologies are the overarching programming philosophy that a

[25] The term theory has its own history and varieties of meaning, but I'm going to use the Merriam Webster dictionary definition, which defines theory as the general or abstract principles of a body of fact. "theory" Merriam-Webster.com. 2011. http://www.merriam-webster.com (July 19, 2011).

programmer uses when writing code – often their languages and tools reflect this philosophy. Programming methodologies guide the way a programmer conceptualizes the program. Programming theory combines these three elements: languages, tools, and methodologies.

I have defined four specific terms that will be used throughout the dissertation:

Complexity in software is the interaction among the people working on the project, the scope and scale of the program, and the algorithmic complexity.

Verification refers to the broader practice of verifying that a program is accurate to its specifications, including both using mathematical proofs to deduce the accuracy of and practical testing techniques.

Programming specialists work on ranges of topics related to the theory of programming, not simply defining or implementing programming languages, but programming methodologies, programming tools, and programming languages. I am defining this set of work to be **programming theory**. Finally, throughout this dissertation I use the concept of a worldview, to encapsulate both the social context and technological changes that create a new branch of programming theory - a methodological shift in the discipline.

The History of Software in the History of Science & Technology

One of the most interesting aspects of researching the history of computing is the role early programming specialists played in their own history. The field regularly engages with questions about what programming is - whether it is a craft or a science, whether it is a sub-discipline of engineering or technology. Early programming specialists don't doubt the importance of the role of computing in society and behave as such - attempting to preserve the history of the discipline while still practicing it. Trade literature like *Byte* and *Datamation* are two such examples, but the *History of Programming Languages* Conferences, and the practitioner founded *Annals in the History of Computing* are unusual because of their focus on the history of the field. Often the focus of this early work is on hardware, particularly on the early developments of the computer. Moreover, because these were written by the participants they are full of firsts and pioneer stories. This reinforces the hype about the computer, portraying it like a train that is barreling towards society and changing it upon impact. Yet, while there is a deterministic feel to these narratives, early computer practitioners have adopted theories from the history of science and technology and applied it to their burgeoning discipline. This is equally obvious in the practitioner-written histories of software.

Thomas Kuhn's theory of paradigm shift and revolution is attractive to the computing community, and even in current marketing of computer products we see this rhetoric. Programming specialists use this theory to talk about changes in software methodology, as in the "structured programming revolution" or the "paradigm" of software engineering.[26] Bjarne Stroustrup's philosophy of programming, which he used to justify the decisions he made during his work on C++, heavily referenced Kuhnian theory.[27] Programming specialists also couch their work in terms of Popper's description of scientific method. They use Popper both as an ideal for the discipline to move towards and as a rule-set that they believe to be following.[28]

Actors in the early history of computing compare their work regularly to the assembly line, referencing the Model T, the American system of mass production and Taylorism.[29] For example, in Michael Mahoney's article "The Roots of Software Engineering," he described a presentation on software by M.D. McIllroy of Bell Telephone Laboratories where McIllroy used metaphor and simile to call to mind

---

[26] P. Wernick and T. Hall, "Can Thomas Kuhn's Paradigms Help Us Understand Software Engineering?" *European Journal of Information Systems* 13, no. 3 (2004), 235-243.

[27] Bjarne Stroustrup, *The Design and Evolution of C++* (Reading: Addison Wesley, 1994).

[28] F. Xia, "Software Engineering Research: A Methodological Analysis," in *Fourth Asia-Pacific Software Engineering and International Computer Science Conference* (APSEC'97 / ICSC'97 1997), 229-236.

[29] Michael S. Mahoney, "Issues in the History of Computing," 1-5.

the machine tool industry. McIllroy describes the division of labor to evoke programming reusable, standardized interchangeable modules. This is actually where the term "software engineering" comes from – it was deliberately chosen to be provocative, implying the need for software to be manufactured, using traditional theoretical foundations and practical disciplines.

However, in 1967, when the term became a part of the common language used to discuss software, no one knew if it was possible to manufacture software. No one knew if there were techniques and practices on which to base this new discipline. This failure of software engineering to conform to the manufacturing metaphor is oft-repeated. [30]

In 1975 Fred Brooks described these failures of software engineering and in 1980 C.A.R. Hoare presented these same failings in his Turing Award Lecture, "The Emperor's New Clothes."[31] Perhaps the failure results in the difference between the metaphor and the reality of programming. In programming the means of production of software remains firmly in the hands of programmers. This differs from the metaphorical assembly line or system of mass production, where the

---

[30] Michael S. Mahoney, "The Roots of Software Engineering," *CWI Quarterly* 3, no. 4 (1990), 327.

[31] Ibid., 328.

company or owner controls the means of production and the workers simply implement the plan.[32] This raises an important distinction for historians of computing: that they should be engaged by the metaphors used by the field, but also to be critical in their examinations of such metaphors.

As Michael Mahoney suggested, it is possible that the reason that the computing community is so enmeshed in the history of technology is because, despite a long prehistory, the computer itself had no inherent precedents. Therefore, the actors involved in the field had to find their own precedents, from their own past history, but also from the history of technology.[33]

Historians of computing use this literature as primary sources, to explore the broader context of computing. Moreover, the history of the computer is a constellation of histories. It is the histories of all of the technologies that make up the computer and the many communities of people that shaped and molded its future. It is within this nexus of histories, written by historians, that I place my own work.

---

[32] Michael S. Mahoney, "Software: The Self-Programming Machine," in *From 0 to 1: An Authoritative History of Modern Computing*, ed. Atsushi Akera and Frederik Nebeker (New York: Oxford University Press, 2002), 91 – 100.

[33] Mahoney, "The History of Computing in the History of Technology," 113-125.

Agents of Change

Technological determinism is an all too popular explanation of how technology changes. It is the theory that technology causes social change, while technical changes are caused by factors internal to the technology.[34] Like many fields, the programming field tends towards technological determinism. Much of the literature in the field implicitly explains software changes as improvements and progress. This change is seen to be organic, without actors, and insulated from the broader socio-economic context. While changes in software design, methodology, and language may result in more economically written programs; this is a convenient by-product of "better" software. This is what Sally Wyatt describes as justificatory technological determinism, in the sense that it is often used by the actors to whitewash over the challenges of innovation to leave a smooth line of progress.[35]

Because computing professionals have been so involved in writing their own history we see numerous articles and books detailing the advances of computing hardware in just these terms, where they begin with the allegorical story of a lone inventor (or small group of

---

[34] MacKenzie, "Marx and the Machine," in *Knowing Machines,* 25.

[35] Sally Wyatt, "Technological Determinism is Dead; Long Live Technological Determinism," in *The Handbook of Science and Technology Studies*, 3rd ed., ed. Edward J. Hackett et. al. (Cambridge: MIT Press, 2008), 174 – 175.

inventors) who unleash a primitive computing technology. This technology then improves to become the recognizable computer part, but this allegory does not detail the continued work of the discipline to refine the invention or discussing the decision making process that molds the technology. The why of computing innovation is often neglected, because the answer is always that this innovation is "better" than the previous advance. This leads to the appearance of inexorable progress towards a platonic ideal of computing.

Historians of computing take a more nuanced approach, examining the web of economic and social relationships and interactions that shape the computing discipline. The increasing interest in software, over the engineering feats of hardware design, has been adopted by historians of computing and more contextual histories that describe the broader environment that both shapes and is shaped by the computer itself, its software and the computing field.

My work is in this tradition. The dissertation demonstrates that changes in programming are a result of two intractable problems in the software field, that of complexity and verification. Initially, this might sound as though I am arguing that the changes in software are the result of only technical problems, deterministic and internal. Instead, through an exploration of these stimuli, I demonstrate that

the discipline negotiates the meaning of complexity and verification. Discourse and debate about the methods of solving these problems is driven by economic and social concerns as often as it is by the technical problems being faced by the programming community.

Programming as Duality

Artisanal/Theoretical Programming

As Michael S. Mahoney wrote, the computer is not just one thing, but many things. He argues that the history of computing has a dual origin, the hardware and the software, which stem from two different disciplines, the history of technology and the history of mathematics, which come together to create the modern computer. I argue that programming is not just one thing.

Within the small community of programming specialists, there are those who pursue the theoretical principles of programming, like computability or formal verification. There are programming specialists working in commercial settings, which we can generalize as pursuing speed and efficiency. But, this is too limiting, it is just as reasonable to expect a programming specialist in the commercial world to be pursuing the holy grail of verifying program correctness, while a programming specialist in an academic setting is pursuing speed. My

work illustrates the tensions, communications, and changing alliances within this community.

This is not a case of science and applied science. Programming specialists are fluid, they move from the academic to the industrial settings. Programming specialists in academic settings are not working in a context free environment, producing theories to be applied by the vocational programmers in the trenches. Nor are commercial programming specialists engaged only in the implementation of academic theories. Many of the large commercial concerns between the 1950s and the 1980s had basic research departments. For example, Xerox had the Palo Alto Research Center (PARC) and AT&T had Bell Labs. This illustrates how programming specialists must work within a continuous feedback loop between theory and practice, in both the academic and commercial settings.

I am not the first historian to note that there are different tensions and divides within the broader software community. For example, Mahoney argues in his article, "Histories of Computing," that there are two disciplines, computer science and software engineering.[36] In his description software engineers are the programmers who work on implementing software applications and

[36] Mahoney, "Issues in the History of Computing," 773.

there are computer scientists who work on the theory of programming. Discourse between these two groups is often a feedback loop, illustrating the shortcomings of the theoretical work in the commercial programming environment or programmers highlighting what would be a useful tool to the computer scientists. In other fields this breakdown is most often academic — in the sense of university researchers, as opposed to those in industry. However, I argue that the divide within the programming specialists is not one of science and applied science, practical versus theoretical, or commercial versus academic. Instead, as I will illustrate in the body of this dissertation, the root of many of the tensions between the 1960s and 1980s is between programming specialists pursuing formal verification techniques and those pursuing a more pragmatic, testing based approach. These tensions drive a number of the significant debates in the literature.

This divide should engage both historians of science and historians of technology. The history of software is not exclusively the domain of the history of technology. Herbert Simon called software a science of the artificial. Software is not natural. Programs exist only because people build computers and write programs for them. However, those programs are mathematical in nature, a fact illustrated by the consistent attempts to mathematically verify the program and

its design — a science of software, if you will.[37] The computer is, in some ways, representing at least some part of the world. And when it runs programs it simulates the world, and, from that simulation, real world decisions are made. In this sense, software functions as science. The computer is an unusual object of historical study: it began as an experiment, it became a tool and from a tool it has become a medium through which we can explore the world. This brings us back to verification and complexity. If we use a program to model the world and base scientific decisions on these models, we must be sure that we have modeled the world correctly, designed the program, and that the program is running accurately. Yet, while we are trying to model small slices of the real world with increasingly more detail, the software gets more complex.

Tacit and Explicit Knowledge

In science and technology there is both tacit and explicit knowledge. Explicit knowledge is information that can be formulated in a transferrable, impersonal medium, like written documents or computer files. Tacit knowledge is knowledge that either has not or

[37] Michael S. Mahoney, "Software as Science – Science as Software," in *History of Computing: Software Issues*, ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur Norberg (Berlin: Springer, 2002), 27.

cannot be explicitly formulated.[38] The records of technology are rarely written documents; the knowledge is often encapsulated in the artifact itself. Thomas Hughes argues that technologists do, they don't write about what they do.

Tacit knowledge is a significant problem in the history of software. I have argued that introducing new people into the programming discipline increased the complexity. Part of the reason for this is that so much of the programming field was built on a foundation of tacit knowledge and assumed experience. With a deluge of new programmers that didn't have that tacit knowledge and weren't physicists, engineers, or mathematicians they also did not have the same background which facilitated communication between programmers. This resulted in increased complexity.

Opening a Black-Box

My work is intended to open the black-box of programming. I explore the way the programming field conceptualizes their programs and how that conceptualization (or world view) changes over time in relation to specific stimuli. I do this by exploring how one would write software in the different programming methodologies and assessing the changes. Looking into the black-box of programming allows us to

[38] MacKenzie, "Tacit Knowledge and the Uninvention of Nuclear Weapons" in *Knowing Machines,* 215.

weigh the often-times contentious debate in the literature about change against the real changes being accepted and rejected in the code. This examination of the technological nuts and bolts separates rhetorical change from real change.

Donald MacKenzie speaks of the importance of opening black-boxed technologies. In his essay, "Nuclear Weapons Labs and Supercomputing," MacKenzie found that because the Nuclear Weapons Laboratories were the primary customers of supercomputers, they influenced the architecture of the computer by emphasizing the importance of floating point arithmetic speed as a criterion of computer performance.[39]

This dissertation strives to make similar use of opening the programming black-box. As Mahoney and Perlis predicted, when we look at the black-box of programming we find that the acceptance and dissemination of certain languages are directly related to economic and social factors.[40] It appears that upon opening the black-box and seeing how the technology is made and stabilized, we find that the inputs and outputs are not simply technical, but are in fact an entire

---

[39] MacKenzie, "Nuclear Weapons Labs and Supercomputing," in *Knowing Machines*, 125 – 129.

[40] Mahoney, "Issues in the History of Computing," 772.

web of social, economic and political concerns that are enmeshed with the technical postulates and assumptions during the creation process.

Using Programs as Historic Documents

Taking programming classes at the University of Sydney in the late 1990s was an exercise in change. Object oriented programming was trickling down into the freshman syllabus, while some professors were holding on to structured programming techniques. During my time in the department I saw a shift in the programming languages used for education purposes, from Prolog to C++ to Java, all within three years. Each of these changes was heralded as revolutionary. During this time I was also taking classes in the history of science and technology. Many of the classes focused on change in science and technology and methods that could be used to explore changes in different fields and disciplines.

The history of physics is one such field that inspired me. In my undergraduate classes I was shown how historians unraveled the changes in the worldview of natural philosophers during the Scientific Revolution. I was specifically influenced by seeing how the change from an earth-centric worldview to the idea of a solar-centric understanding of the world could be seen by contrasting the internal work of natural philosophers with the external scientific discourse of

the time. I believe that seeing this interplay of the internal and external discourse of the field brings both the history and the science of the time alive. That the internal workings of science could be influenced by the socio-economic-political milieu that scientists were working within was, for me, a new concept. I wanted to transfer that experience to a field that I find intriguing – the history of computing.

During my research, I came across two anecdotes that were influential on this dissertation. Coincidentally, they are also anecdotes that influenced Mahoney. The first is an anecdote related by Alan C. Kay, the creator of Smalltalk. In this story he walked into the University of Utah and found on his desk a memo asking him to make ALGOL for the Univac 1108 computer work. He describes the process of unrolling the program listing down the hall and crawling over it and reading the code. In doing so, he discovered how ALGOL worked.[41] Another story, one from Tracy Kidder's book, *The Soul of a New Machine*, described Tom West (a project leader at Data General) looking at the circuitry of Digital Equipment Corporation's VAX machine and seeing a reflection of the corporate organization it evolved

[41] Alan C. Kay, "The Early History of Smalltalk," in *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993* eds. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., (New York: ACM, 1994), 516.

within.[42] If Alan Kay could recreate how the ALGOL programming language worked by reading it as a document, then did this mean that all object code was essentially a document? And, if company organization could be found in the hardware of a minicomputer, why couldn't the social and economic milieu be found in the object code of a software application? Mahoney was also influenced by these vignettes and came to similar conclusions. He writes about the concept of "reading the products of practice," like object code, arguing that it could be a historically interesting theory.[43] Mahoney specifically discusses the importance of looking at programs – and the difference between what a program does and what its documentation says. He talks about the importance of capturing the tricks of the trade, the tacit knowledge that was described above. For historians, to capture those tricks could be through reading object code as a document. This is what I have attempted to implement.

In this dissertation I have treated software as a document, one that could be read and understood, with the hope that it would demonstrate the changes in the way programmers wrote code between the early days of assembly language and the sophisticated

---

[42] Tracy Kidder, *The Soul of a New Machine* (Boston: Little, Brown and Company, 1981), 32 – 33.

[43] Mahoney, "Issues in the History of Computing," 772-781.

programming environments of object oriented languages. The problem with this is that, in the later years of the history, the "source code" (the program written in human readable symbolic format) is almost always proprietary to the company that created the software. Finding a stable software application that changed over time, but that would be freely available as an object of research, was almost impossible. The best lead on this kind of work that I uncovered was the source code for games. In the earliest years of game coding, the game's source code was typed up in computer enthusiast and hobbyist magazines. However, by the time of hand-held consoles, these were few and far between, because as the games grew more lucrative their secrets became more protected. This is a problem pointed out by Mahoney, when he argues that much of the art of programming is undocumented, and the software is becoming inaccessible. Furthermore, the decisions made when creating those programs are buried in corporate records.[44]

As a result of these difficulties, I decided to focus on changes in programming languages. This made it much easier to track because programming languages by their very definition must be transparent to the user. Instead of using applications written by programmers, I

[44] Ibid.

wrote small programs in the style of the different languages or using pseudo code to illustrate the different methodologies, basing them on templates from well-known programmers, like Donald Knuth. I used these anecdotal programs to uncover changes in the way programmers write software and then used the literature of the programming field to contextualize and situate these changes in the broader web of the socio-economic milieu of the programming field.

My hope is that this use of software as a primary source, a document that can be used to read human motivations in technical things, will be a useful perspective for others interested in the role of software in society. I also hope that this serves as an impetus to archive object code for future generations.

Historiography

In the past thirty years, the history of software has become a flourishing sub-discipline of the history of computing.[45] I see my work as one part of this larger puzzle. That puzzle encompasses the entirety of software history – including the history of the software industry, professionalization, the relationship between hardware and software, and the history of software applications. This section will place this dissertation into that broader context.

[45] I take this from the first publication of Annals of the History of Computing (1979) and the founding of the Charles Babbage Institute (1978)

In a general sense, much of the work in the history of software parallels my dissertation chronologically. In this dissertation, I have examined the changes in programming practice and theories over a (comparatively) long time period — the 1950s through the 1980s. Much of the work in the history of software (and in the history of computing more broadly) centers upon the pre-history, emergence and early years of the software field, between the 1950s and early 1970s.

There are six authors that intersect with this dissertation significantly. Michael Mahoney's work has steered much of this dissertation. The role of software as a science is the presumption this dissertation is based upon.[46] As a result, much of the body of the dissertation uses concepts from the history of science as frequently, if not more frequently, than concepts from the history of technology. The idea that there are a number of histories that make up the broader history of software has further influenced this dissertation.[47] The concept prompted my dynamic definition of programming specialists, who move in and out of the narrative, as they move around within the larger software field. Finally, Mahoney's discussion about the difference between what software is said to do, and what

---

[46] Mahoney, "Software as Science – Science as Software," 27.

[47] Mahoney, "Issues in the History of Computing," 773.

the software actually does, in conjunction with his discussion about reading the products of practice, influenced my methodological decision to use source code as a document throughout this dissertation.[48]

Stuart Shapiro's dissertation, *Computer Software as Technology: An Examination of Technological Development*, intersects directly with my work. While our actors, timeline, and sources often overlap, our interpretation and narrative is very different. Shapiro's starting point for his interpretation is that software is a technology, and a technology unlike other technologies – one that is in crisis. Alternatively, I argue that software has elements of both a technology and a science. This allowed me to use traditional history of science and history of technology methods in my analysis. Moreover, unlike Shapiro, I have shifted the focus of the narrative from the insider's account, which centers on crisis and revolution, by concentrating on the technical changes in the discipline.

Shapiro and I also differ in our organization. Shapiro chose to organize his work chronologically, creating a number of periods in software. By organizing my dissertation around the three styles or

---

[48] Ibid., 772-781.

methodologies of programming that dominated the landscape during the time period of interest, I was able to illustrate connections between different languages, and different methodologies, despite the fact that they did not overlap chronologically. This organizational choice also provided a selection criterion to guide which programming languages to include in the dissertation. I have chosen to focus on languages that were emblematic of the methodologies (often considered a style of programming) of the time or had a particular significance in the rise or fall of such a methodology.

My dissertation does overlap significantly with Shapiro's work. Our time periods are similar, although mine extends further into the 1980s. The same actors appear in both of our narratives. Shapiro even discusses the problem of complexity in software and the debates over verification, but he defines this as a part of the structured programming debate.[49] I see complexity and verification as stimuli for change throughout the history of software, not problems to be overcome during a particular debate, and as a result, I trace a different path in my dissertation and provide a different narrative, particularly in the shift from ad hoc programming techniques to structured programming.

---

[49]Stuart S. Shapiro, *Computer Software As Technology: An Examination of Technological Development,* (Pittsburgh: Carnegie Mellon Press 1990), 111-152

I disagree with Shapiro's assessment that structured programming arose as a direct result of the software crisis. Chronology itself refutes this argument, as Dijkstra's work on structured programming began prior to the 1968 conference that computer scientists and historians alike ascribe as the beginning of the move to software engineering.

Like Shapiro, Peter Mark Priestley's recent dissertation "Logic and the Development of Programming Languages, 1930–1975" intersects directly with my work. However, Priestley's work has an issue similar to Shapiro's dissertation. Priestley's work is chronologically organized and talks exclusively about how programming languages change.[50] My dissertation differs significantly in two ways. First, I focus not on programming languages, but programming methodologies, which brings this dissertation explicitly into the realm of theory. Moreover, I am addressing why programming theories changed, and arguing that this is directly related to two specific stimuli, verification and complexity. This allows me to justify which languages I look at, as emblematic of their programming methodology. Second, on a more practical level, I argue that this dissertation extends the field of the history of software, by

[50] Peter Mark Priestley, "Logic and the Development of Programming Languages, 1930 – 1975" (Ph. Diss. University College London, 2008)

focusing on the changes occurring between the late 1960s through to the 1980s. Significantly, this is the first historical work to address, in depth, the shift to object oriented programming.[51]

Donald MacKenzie's work, *Mechanizing Proof* provides an excellent introduction of the importance of verification in computing. His essays illustrate the importance of trusting the computer in our society, referencing historical works on the subject including Theodore M. Porter's *Trust in Numbers*, Sherry Turkle's *The Second Self: Computers and the Human Spirit*, and Steven Shapin's, *A Social History of Truth: Civility and Science in Seventeenth-Century England*. In the essays, MacKenzie details the emergence of the computing communities focusing on program correctness, drawing connections between program verification, mathematics, and artificial intelligence.[52]

Thomas Haigh's work is one that I would define as a parallel narrative to my own story, focusing on the communities that emerge and decline with the advent of computing, during the shift from office machines to electronic digital computers. Haigh's work describes

---

[51] Priestley's dissertation has recently been published, but I have yet to get a copy of the book: Peter Mark Priestley, *A science of operations: machines, logic and the invention of programming,* (New York, London: Springer, 2010)

[52] Donald MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust* (Cambridge: MIT Press 2004), 57.

different approaches to the adoption of computing technology in related fields, focusing on the dichotomy of those that failed to accept these changes and those that embraced the computer as first adopters.[53] Haigh has defined this community of specialists as managerial technicians. Haigh illustrates how these managerial technicians attempted to separate the technical administrative and systems based techniques from management, creating an additional professional branch in corporate America. The focus of Haigh's dissertation is not only the computer, he has incorporated the computer into a broader business history – telling the story of the design, construction and control of the large scale systems that have become routine in corporate administration. Like Nathan Ensmenger, he focuses on the professionalization of computer specialists, however, he tells a longer story, of the existing corporate administration specialists who were in these positions prior to the computer. Ensmenger's story focuses on that of vocational programmers who grew up with the computer.

Nathan Ensmenger's book, *The Computer Boys Take Over*, also explores the professionalization of software creation.[54] His book traces the history of the community he calls "the computer boys." These

[53] Haigh, "Technology, information and power"

[54] Ensmenger, *The Computer Boys Take Over*

computer boys differ from Haigh's existing managerial technicians, because they are the generation that grew with the computer, who must then establish their own professional roles within traditional hierarchies. Chronologically, the story that Ensmenger tells starts, like mine, with the advent of the computer. His community is made up of vocational programmers, who are directly influenced by the trends and fashions coming out of the programming specialist community that I am concentrating on.

While our work intersects chronologically, my study differs from Haigh and Ensmenger's studies. Our community of focus is different – Haigh is dealing with managerial technicians who are, essentially, the computer users in my story. Ensmenger is focusing on the more pragmatic, vocational programmers. These vocational programmers are users who are implementing applications designed by the programming specialists that I focus upon. I see our work as parallel histories of different parts of the software field.

A number of works in the history of computing contribute to the understanding of the early history of computing. Paul Ceruzzi's, *The Reckoners* explores the pre-history of the computer.[55] Ceruzzi describes the four major developments that led up to the digital,

[55] Paul Ceruzzi, *The Reckoners* (New York: Greenwood Press, 1983)

electronic computer: Konrad Zuse's German Z3, Howard Aiken's work on the electromechanical Mark I, the work at Bell Labs, and ENIAC. Nancy Stern's work, *From ENIAC to UNIVAC* explains how the digital computer moved from an experiment, to a tool, to the foundations of a new industry.[56] Arthur Norberg's work, *Computers and Commerce*, describes the role of two of the earliest computer manufacturers, Engineering Research Associates (ERA) and Eckert and Mauchly Computer Corporation (EMCC) and their eventual amalgamation into the larger Remington Rand.[57] David Alan Grier's work "The Eniac, the Verb 'to Program' and the Emergence of Digital Computers" illustrates where the verb to program came from and defines what it meant in these early years of programming.[58]

Atsushi Akera and Mitchell Marcus' work on the architecture of the ENIAC, "Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture" is an important technical history of the machine and how the ENIAC

---

[56] Nancy B. Stern, *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers* (Bedford: Digital Press, 1981)

[57] Arthur Norberg, *Computers and Commerce: A Study of Technology and Management at Eckert-Mauchly Computer Company, Engineering Research Associates, and Remington Rand, 1946-1957 (History of Computing),* (Cambridge: MIT Press, 2005)

[58] David Allan Grier, "The Eniac, the Verb 'to Program' and the Emergence of Digital Computers," *IEEE Annals of the History of Computing* 18 no. 1 (1996).

functioned.[59] Kurt Beyer's dissertation, *Grace Hopper and the Early History of Computer Programming, 1944-1960*, went into significant detail about the way early computers were programmed, through the eyes of Grace Hopper.[60] Martin Campbell-Kelly's "Programming the EDSAC: Early Programming Activity at the University of Cambridge"[61] and his dissertation *Foundations of computer programming in Britain, 1945-1955* illustrate the realities of early programming.[62] The dissertation, in particular, illustrated clearly how programming was conducted.

Between the emergence of electronic, digital computing in the late 1940s and the adoption of automatic coding systems in the mid1950s, the computer was transformed from a laboratory machine to a commercial tool. The history of the software industry bears this out, illustrating the increase in computing projects and software firms during this time period. Jeffrey Yost's work, *The Computer Industry*, and Martin Campbell-Kelly's, *From Airline Reservations to Sonic the*

---

[59] Atsushi Akera and Mitchell Marcus, "Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture," IEEE Annals of the History of Computing 18 no. 1 (1996): 17-24.

[60] Kurt Beyer, "Grace Hopper and the Early History of Computer Programming, 1944-1960," PhD diss., University of California at Berkeley, 2002; Campbell-Kelly, "Programming the EDSAC."

[61] Martin Campbell-Kelly, "Programming the EDSAC: Early Programming Activity at the University of Cambridge," *IEEE Annals of the History of Computing* 2 no. 1 (1980): 7.

[62] Campbell-Kelly, Martin, *Foundations of computer programming in Britain, 1945-1955*, Ph.D. diss., Sunderland Polytechnic, England, 1980.

Hedgehog, illustrate the corresponding business changes that support this transformation.[63] Campbell-Kelly's study of the software industry illustrates the rapid pace of the changes in commercial programming, from custom programming to the off the shelf software. Along with this history of software, he details the changes in computers themselves, from mainframes to personal computers.

Of particular influence in my work are sources that define or contextualize programming languages. For any aspiring historian of software, these articles provide both technical detail and, when read together, can illustrate a narrative. The 1984 special issue of *Annals* devoted to FORTRAN contained a number of influential articles, including Robert A. Hughes article, "Early FORTRAN at Livermore" and J.A.N. Lee's "Pioneer Day."[64] D. Nofre's work on ALGOL, "Unraveling ALGOL: US, Europe, and the Creation of a Programming Language"[65] juxtaposes a language created by international committee with Jean Sammet's domestic committee in "The Early History of COBOL." Arthur Norberg and Judy O'Neil's, *Transforming Computer Technology*,

---

[63] Campbell-Kelly, Martin, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge: MIT Press, 2004, 29-53; Jeffrey R. Yost, *The Computer Industry*, (New York: Greenwood Press, 2005)

[64] Robert A. Hughes, "Early FORTRAN at Livermore," in "FORTRAN's Twenty-Fifth Anniversary," ed. J.A.N. Lee and H.S. Tropp, *IEEE Annals of the History of Computing* 6 no. 1 (1984) and J.A.N. Lee, "Pioneer Day," *IEEE Annals of the History of Computing* 6 no. 1 (1984)

[65] D. Nofre, "Unraveling ALGOL: US, Europe, and the Creation of a Programming Language," *IEEE Annals of the History of Computing*, 32 no. 2 (2010)

describes LISP, a language that has only a few devoted followers, but transformed the programming landscape.[66]

However, there is still much information to be found in what would ordinarily be called primary sources that is difficult to tease out of the literature in the history of computing. The ACM SIGPLAN conference *History of Programming Languages*.[67] The HOPL proceedings are an interesting mixture of primary and secondary literature. While the articles were written by practitioners, they are reflections on their own past. These articles delve into the truly technical details, in a way that the secondary literature does not (and should not).

The idea of the software crisis and the shift to structured programming is one of the most interesting episodes in the software field to date. It was a very public, but theoretically driven debate, and paralleling significant shifts in other scientific fields. Nathan Ensmenger speaks about the role of the perpetual software crisis, arguing that it is a byproduct of the continued "jurisdictional struggles" that occur during periods of professionalization, and linking that

---

[66] Arthur Norberg and Judy O'Neil, *Transforming Computer Technology: Information Processing for the Pentagon*, (Baltimore: Johns Hopkins University Press, 1996)

[67] Richard L. Wexelblat, ed., *HOPL-I: Proceedings of History of Programming Languages – I* (Blue Bell: Sperry Univac, 1981)

definitively to the issues that arose in the 1960s.[68] While, I agree that the discourse surrounding the 1960s software crisis has at its core an issue of professionalization, I argue that the theoretical issues at hand were lost in the rhetoric of crisis. I illustrate a complementary story that focuses on internal changes in the discipline, particularly the structured programming revolution, that are perceived as resulting from the frank conversation in the 1968 conference, were already being pursued and were driven by stimuli internal to the software community — complexity and verification.

The shift to the object oriented programming methodology has not, yet, been well covered by the historical literature. Nathan Ensmenger mentions object oriented programming briefly, describing Brad Cox's early promotion of the methodology as a silver bullet for resolving complexity, but for the most part, the historical literature is silent on the topic of object oriented programming.[69] A new scholar, Hansen Hsu has recently begun working on object oriented programming as part of his broader ethnographic study on Apple, but his manuscripts are, at this point, unpublished.

---

[68] Ensmenger, *The Computer Boys Take Over.*

[69] Ibid., 108

There are also journalistic accounts of the advances in software. Outside of Tracy Kidder's *Soul of a New Machine*, an important work that raises many questions about the relationship about business organization and computer architecture, to name but one of the themes. There are also current journalistic accounts that detail both technical and industrial changes. For example, the technical details of current programming paradigms are often most available in trade literature publications like IEEE's Computer or Code Magazine.

Chapter Outline

The second chapter of my dissertation lays the foundation for my thesis that programming methodologies change in response to two specific stimuli, complexity and verification. It illustrates how programming was originally conducted, to contrast it with later methodologies. This early programming era saw the adoption of three important innovations. Programmers moved from the direct manipulation of binary, hexadecimal or octal code to more sophisticated symbolic manipulation, through the adoption of three major technological innovations: the stored program, sub-routines, and symbolic notation. I argue that even at this early moment, these changes were in response to the rise of a fundamental problem in computing: complexity, or the proliferation of people and methods.

Prior to 1954 almost all programming was done in machine or assembly language. The third chapter of my dissertation focuses on the shift from assembly language to automatic coding systems – higher level languages with a more English-like syntax. I illustrate that the adoption of automatic coding systems was not automatically accepted by the field, and instead, that there was significant debate about the necessity of this type of tool in the nascent computing industry. Yet, by 1960 automatic coding systems, specifically FORTRAN and COBOL were widely accepted to be the future of the programming community. Automatic coding systems introduced a new kind of complexity and new problems of verification, by introducing a new level of abstraction away from the machine.

The fourth chapter of my dissertation covers the period of the structured programming "revolution." While structured programming is often called revolutionary by programming specialists, I will argue that the change from the ad hoc programming style of the early years of computing is a more subtle shift than the traditional narrative suggests. I further argue that structured programming arose as a response to vocal complaints in the programming community. This chapter opens by showing that on the surface these complaints sound vague, but further analysis demonstrates that there are two specific problems at the root of this discontent in the programming

community: complexity, or the proliferation of people and methods; and verification, the (in)ability to verify that a program functions as intended. The chapter demonstrates that structured programming was developed in response to the problems of complexity and verification and became popular because it claimed to solve these problems. While structured programming alleviated some of the tension created by complex, inaccurate programs it was not a silver bullet that solved those problems, and as a result fell out of favor.

Structured programming was replaced by object oriented programming as the dominant programming methodology in the 1980s. My fifth chapter addresses this shift. While I have argued that structured programming was not revolutionary, I argue that object oriented programming was a significant change in the way programmers perceive software. I begin by demonstrating that the origins of object oriented programming were concurrent with the origins of structured programming. Both structured programming and object oriented programming were in response to similar concerns about complexity and verification. Unlike structured programming, object oriented programming was accepted gradually and was only fully accepted with the distribution of the C++ programming language, which provided a commercially viable vehicle for the object oriented methodology. This chapter explores the origins, evolution and

acceptance of object oriented programming and the roles complexity and verification played in the movement from structured programming to object oriented programming.

The final chapter of my dissertation addresses the changes that occurred in programming theory after the adoption of object oriented programming. After the mid-1980s, plurality became the dominant theme in the software field. This doctrine of plurality superseded the community's previous goal of finding a single universal solution. We see this change in direction in methods of verification, styles of programming languages, and new methodologies - like concurrent programming.

Conclusion

This dissertation is a case study of complex, technological change. I focus on the technical changes in the methods, designs, and languages programming specialists use to write software: what those changes look like, why they happened, and their ramifications. I argue that software has not proceeded down a fixed evolutionary, but that there are instead many technical and social stimuli that drive technical changes in software design, methods, and languages. I have focused on two of these technical stimuli and the discourse surrounding them - complexity and verification

Chapter 2: A Pre-history of Programming

The Origins of Modern Programming

Early Computing

In 1943, John Mauchly and John Presper Eckert, Jr. began work on what would be the first electronic, digital computer at the Moore School of the University of Pennsylvania. Their work was supported by the U.S. Army's Ballistics Research Laboratory. The purpose of the ENIAC (Electronic Numerical Integrator And Computer) was to compute ballistic firing tables — an urgent requirement during World War II. World War II was a time of significant change in weaponry, many of which required firing tables. Ballistic firing tables furnish gunners with the information they need to accurately fire weapons. These ballistic firing tables are not a trivial mathematical undertaking; they depend upon the applied mathematics discipline of exterior ballistics, which incorporates atmospheric, speed and distance variables using numerous non-linear, differential equations to direct gunners to properly target the weapon. At the time, these tasks were completed by many (often female) human computers, but there was a significant lag providing firing tables for new artillery.[70] The ENIAC was

[70] Stern, *From ENIAC to UNIVAC,* 12-14.

conceived as a way to solve this bottleneck by solving equations at electronic speeds.

The ENIAC machine was originally thought of as an electronic equivalent to a desk calculator and used many elements of tabulating machines. Data was input and output using punch cards, while the instruction set was "programmed" using a wired plug board and, later, portable function tables.[71] We can see in Figures 2 and 3 that "programming" the ENIAC is different from our modern conception of programming. Programming the ENIAC was an exercise in circuitry and changing the flow of electricity to produce a desired result. [72] As a result, the women who were first tasked with setting up the circuitry for the ENIAC were called "coders", while those tasked with organizing the problem were described as "planners." This separation of terms illustrates the perception that this technical reconstruction of the problem was seen as clerical work, like that of punch card operators.

[71] These were developed by the United States Army Ballistic Research Lab at Aberdeen, MD.

[72] Stern, *From ENIAC to UNIVAC*, 24-47.

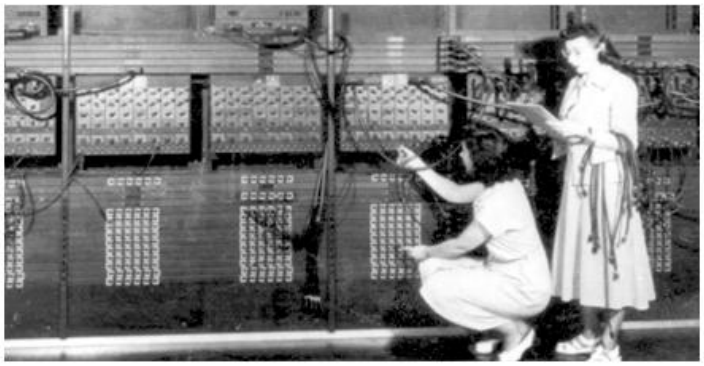


Figure 1: Wiring the ENIAC with a new program.
U.S. Army Photo: from the archives of the ARL Technical Library.
Standing: Ester Gerston; Crouching: Gloria Ruth Gorden.

Despite the differences in the way people programmed, the term "programming" did stem from the work conducted on the ENIAC. Researchers working on the ENIAC wanted to delineate the difference between the electronic computer and the human computer. Human computers used the term "plan" to describe their computational process. The term "programming" was actually repurposed from two different fields: electrical engineering and the military. In electrical engineering the term "program" had been appropriated and expanded from its original use to describe a descriptive notice (like a program for a concert). The term was first applied to the radio (radio programs were broadcast), and then to describe electronic signals carried by a circuit. In the military, the term "program" was employed to mean assemble or organize. Throughout the work on the ENIAC the terminology is slippery, but by the end of the project, the words

"coding" and "programming" were entrenched as the dominant terms within ENIAC.[73]

Figure 2: U.S. Army Photo of the ENIAC
Courtesy of Harold Breaux.
Soldier (CPL Irwin Goldstine) in the foreground is at a function table.

As different as the ENIAC programming environment is from the modern conception of programming, the ENIAC was a programmable machine, separating it from traditional analog machines. Traditional analog machines modeled one function that can be applied to many different data sets. The ENIAC was able to solve many different types

[73] Grier, "The Eniac, the Verb 'to Program'. Moreover, in 1950, when Eckert and Mauchly Computer Corporation was hiring people to program the machines, they were definitively described as programmers in the hiring contract, confirming that within the ENIAC circle these terms had become ingrained, while at GE this same job was titled as a procedures analyst, suggesting a distinction between computer manufacturers and their clients – Oral History, Univac Conference, CBI 200, p. 101 - 103

of functions; in other words, it is a general purpose machine —one that needed software.[74] The need to input the functions into the ENIAC is the birth of the modern programming field.

The functional element of the ENIAC design that sets it apart from traditional, analog machines was its ability to execute loops and store the results of its calculations for use in later calculations. The ENIAC had a central control unit. The control unit (called the master programmer) directed the electronic channels that sent program signals down the program cables. The computing units processed these signals and generated a completion signal, which then prompted other units to begin their operations. This is what makes the ENIAC a general purpose machine.[75]

However, despite oft-repeated claims[76] that the ENIAC was able to execute conditional branching, there is significant evidence that this was an afterthought, provided by one of the later members of the ENIAC team, after the device was in use and stored program computers had moved to the forefront of computing.[77]

---

[74] Ceruzzi, *Reckoners,* 110. Note that the ENIAC was not easy to "re-set".

[75] Grier, "The Eniac, the Verb 'to Program'," 51-55.

[76] Grier's article, Ibid., claims that the ENIAC completed conditional branching, as does the Encyclopedia Brittanica, Online

[77] Atsushi Akera and Mitchell Marcus, "Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture," *IEEE Annals of the History of Computing* 18 no. 1 (1996): 17-24.

Conditional branching is the basic building block of the programmer.[78] With conditional branching, the computer is said to be data sensitive. The program can make decisions based on the results of earlier decisions. So, for example, if a programmer tells the computer to do "A" and then do either "A" or "C" depending on the value of "A", it has made a conditional branch.

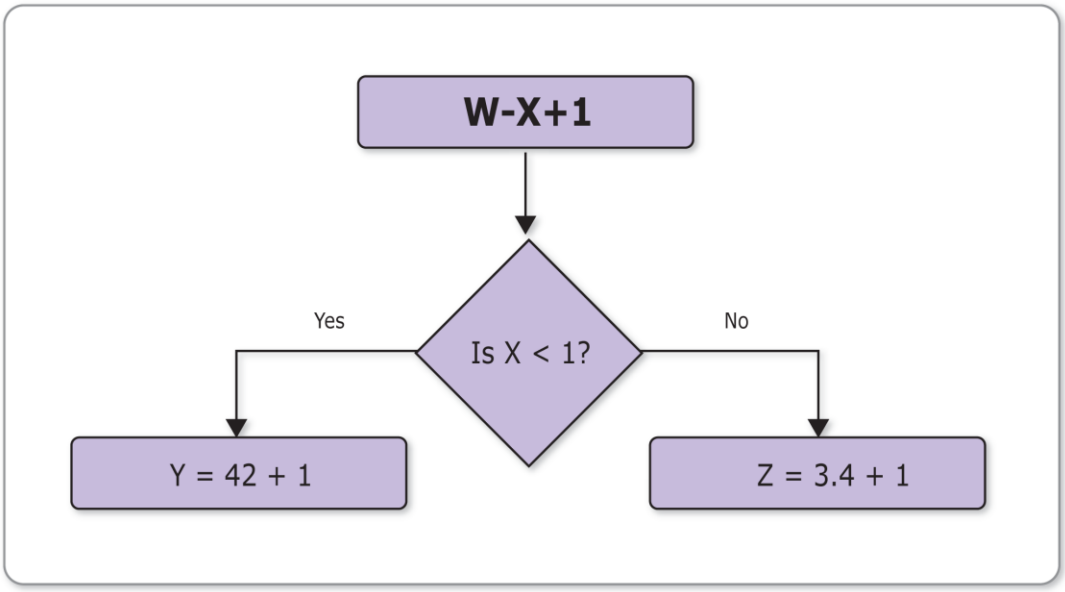


Figure 3: Branching

This is why, even before the completion of the ENIAC, the Moore School team had begun designing its successor, the EDVAC (Electronic Discrete Variable Automatic Computer). Mauchly and Eckert were the

[78] Throughout the dissertation I am using programmer anachronistically, but also generically, to describe how any programmer (whether they are vocational programmers or programming specialists) would carry out a particular operation.

impetus behind the ENIAC, as was Herman Goldstine, who secured much of the funding for the ENIAC. Goldstine, after a chance encounter, brought John von Neumann into the project. Mauchly, Eckert, Goldstine and von Neumann worked in parallel to design the EDVAC. The EDVAC's design was intended to improve on the efficiency and reliability of the ENIAC. Specific difficulties that were identified in the ENIAC included the complexities of changing the programming once the ENIAC was set up to solve a specific problem and the need for conditional branching.

It is remarkable that even at these origins of the computer and programming theory, complexity was a catalyst for change. The catalyst for the move towards von Neumann architecture was to simplify the use of the computer through software, as opposed to the wired plug boards of the ENIAC. The complexity of software may have changed in style, but functionally the problems occurring at this early time have many parallels to current debates about complexity and verification. The physical changes that needed to be made to the plug boards of the ENIAC were intricate. Human error could cause reliability issues. The programs were not reusable, because to use a previous program, the entire plug board would need to be rewired. This rewiring work was time consuming - sometimes it took several days to reset the computer to solve the next function. This creeping

complexity and uncertainty about correctness was the impetus for the design of modern computer architecture, embodied in the design of the EDVAC.[79]

Stored Program Concept

The critical difference between the plug board programming of the ENIAC and the programming of the EDVAC (and other stored program style computers) is the introduction of the stored programming concept. A stored program is exactly what it sounds like: a program stored in the memory of a computer. To implement the stored program concept, computers must be able to take a set of instructions, the program, and store them in a memory device. The computer can then execute the program when required.[80] Without memory devices like the mercury delay lines of the UNIVAC (the UNIVersal Automatic Computer) or the magnetic core memory of the Whirlwind, the stored program concept could not be realized. The ENIAC did not have this facility. The (mostly female) operators of the ENIAC used techniques like that of a plug board telephone system to

---

[79] Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* 2nd ed. (Boulder: Westview Press 2004). See also Paul E. Ceruzzi, "Crossing the Divide: Architectural Issues and the Emergence of the Stored Program Computer, 1935-1955," *IEEE Annals of the History of Computing* 19 no. 1 (1997): 6.

[80] James W. Cortada*, Historical Dictionary of Data Processing Technology,* (New York: Greenwood Press, 1987), 97-98.

change the programs. Alternatively, the EDVAC was designed to use a memory device.

There is controversy over the collaborative effort to produce the design of the stored program computer. However, it has become common to credit John von Neumann as the impetus behind the design of the stored program computer. This is reflected when we describe an electronic, stored program computer as having von Neumann architecture. The stored program concept is not only the guiding principal of computer architecture; it is also the origin of modern programming.

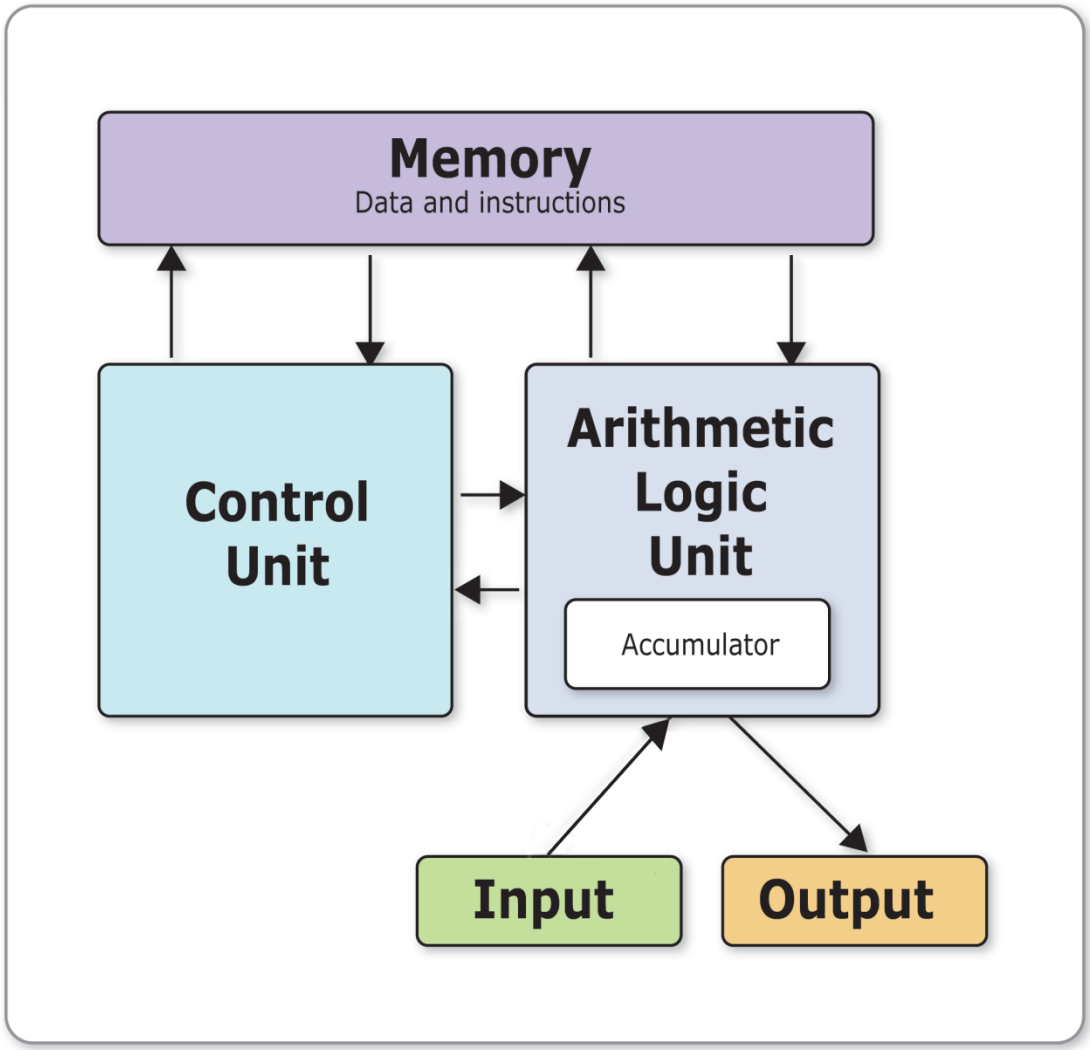


Figure 4: Stored Program/von Neumann Architecture

von Neumann's stored program concept encapsulates a basic architecture upon which all modern computers have been based.[81] This concept treats data and instructions identically. To implement

[81] After the explosion of computing devices following World War II, the reasons and timing of the design of the EDVAC became controversial because of patent claims. John von Neumann's "The First Draft of a Report on the EDVAC," which listed him as the sole author, despite evidence that some of the concepts had originated in a team setting, fueled this fire. This is described in Cortada, *Historical Dictionary*, 190

von Neumann architecture the computer requires serial processing techniques[82] and hierarchical memory.[83]

Treating Data and Instructions Identically

> Any device which is to carry out long and complicated sequences of operations...must have a considerable memory…. it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible.
>
> John von Neumann, Report on the EDVAC, p.3

Von Neumann's most important contribution to computing is the understanding that although we imagine the instruction set (software) to be different from the data it is acting upon, it is the same thing and we can use a single storage medium for both programs and data. This has specific practical benefits.[84] By storing the instruction and data in the same physical storage medium the bottleneck of mechanically inputting data, which was then processed at electronic speeds, was resolved, because data could also be input at electronic speed. It also resolves the problem of re-programming, by storing the instructions electronically instead of using a pre-programmed plug board to store the instructions. The most important contribution though, is that

---

[82]I'm referring here to the decision to execute the programs sequentially. At the time there was a push to also handle the numbers sequentially, a completely separate design feature which was eventually discarded.

[83] Ceruzzi, "Crossing the Divide," 6.

[84] B. Randell, "The Origins of Computer Programming," *IEEE Annals of the History of Computing* 16 no. 4 (1994): 13.

when data and instructions are seen as the same, then the instructions can be changed dynamically, based upon the results of the transformations that the data underwent. This sophisticated use of branching is the basis of modern programming.

Hierarchical Memory

Hierarchical Memory is the layering of computer storage. In von Neumann's design the high capacity, slow storage complements faster, lower capacity storage. For example, in the UNIVAC machine the mercury delay lines would be the fast, lower capacity storage complemented by high capacity, slow retrieval, magnetic tape. This is the equivalent to random access memory (RAM) complementing the hard drive in personal computers of the 1980s through the present day.

Sequential Processing

> The device should be as simple as possible… This can be achieved by never performing the two operations simultaneously.[85]
>
> John von Neumann[86]

The basic cycle of a von Neumann computer is to transfer an instruction from the memory to the processor and execute the instruction. Once the processor executes the instruction, it repeats

---

[85] John von Neumann, "The First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing* 15 no. 4 (1994): 8.

[86] Ibid.

this process. It does not execute several commands at the same time.[87] This differs dramatically from the punch card tabulating machines. In punch card systems, each operation was performed on the entire batch of cards before moving on to the next operation.[88]

After the Stored Program Concept

The Moore School Team of Mauchly, Eckert, Goldstine and von Neumann fell apart after the design of the EDVAC. Mauchly and Eckert were interested in pursuing the commercial applications of the computer, while Goldstine and von Neumann were interested in disseminating these details to a broad academic audience. Eventually, Eckert and Mauchly left the Moore School to form the Eckert-Mauchly Computer Corporation in 1947. The EMCC delivered the UNIVAC 1 to the US Census Bureau in 1951. von Neumann and Goldstine also left the Moore School for the Institute of Advanced Study at Princeton, heading up the IAS computing project. The Moore School team's work on the stored program computer was disseminated widely both through von Neumann's report, "First Draft of a Report on the EDVAC,"

---

[87] This differs from serial or parallel arithmetic, a choice von Neumann remained silent about. In serial arithmetic the processor acts on a single digit, while in parallel arithmetic the processor acts on a whole number (regardless of what number system is being used). In "The First Draft of a Report on the EDVAC" von Neumann did not clearly delineate whether a computer should process everything one bit at a time or as an aggregate of digits. However, consensus later developed over the use of a "word" (an aggregate of binary digits) which would be operated upon.

[88] Ceruzzi, "Crossing the Divide," 9.

and the 1946 Moore School Lectures, "Theory and Techniques of Design of Electronic Digital Computers". The lectures described the concepts formulated during the design of the EDVAC to many of the luminaries in the burgeoning computing field, speeding up the adoption of the stored program model and touching off a period of rapid growth. This forever muddied the waters of priority, later leading to a series of patent lawsuits.[89]

The stored program concept substantially changed the direction of both hardware and software. 1949 marked the development of the EDSAC (the Electronic Delay Storage Automatic Calculator) at the University of Cambridge Mathematical Laboratory by Maurice Wilkes. This was the first practical stored program computer, programmed using punched tape, not unlike the early automatic tabulating machines. Another early machine of note was the Whirlwind Project at MIT, run by Jay Forrester. The Whirlwind project was the basis for the later SAGE (semi-automated ground environment) and the origins of magnetic core memory. Jeffrey Yost argues that Whirlwind was responsible for the momentum that established MIT as an essential center of computing research. [90] At this time Engineering Research

---

[89] Stern, *From ENIAC to UNIVAC.*

[90] Yost, *The Computer Industry,* 30

Associates (ERA) was working on magnetic drum storage, a project that became a complete stored program computer – the Atlas 1.[91]

ERA began as a code-breaking division in the US Navy during World War II. After the war ended, William Norris (who later became the CEO of Control Data Corporation) led much of the team to form a commercial venture. Much of the firm's early work was creating code-breaking machines for the US Navy, but they had significant success with the use of magnetic drum memory. It was also during this time period that Seymour Cray began his career with ERA. Cray is one of the most famous computer architects, capturing the imagination of the country with Cray supercomputers in the 1970s. By 1952, as a result of legal battles and poor sales, ERA's financial prospects were poor. In 1952 Remington Rand bought ERA. This acquisition is particularly interesting, because Remington Rand had already purchased EMCC in 1950.[92] For some time, the two companies operated as independent divisions within Remington Rand, but in 1955 Remington Rand merged with Sperry (becoming Sperry Rand) and ERA and EMCC were merged into one division at that time. The power struggles from this merge prompted Norris to leave, with many of the original team (including

---

[91] Norberg, *Computers and Commerce*, 69.

[92] Ibid., 213

Cray) to create the Control Data Corporation, an influential player in the later history of computing.[93]

It is arguable that the first ERA computer, Atlas, and produced for the Navy, was the first practical stored program computer. Atlas came online in 1950. ERA later sold this computer model commercially as the ERA 1101. ERA's many technical successes, combined with the many leading figures that emerged from the firm, influenced the computing field for decades after the company's inception, particularly in storage techniques.[94] It can be argued that the UNIVAC machine had the most impact on the programming field. This is borne out within this dissertation – as I will show, some of the most pivotal programming advances were conducted on the UNIVAC machine. Holberton wrote her sort programs and Hopper her compilers for the UNIVAC machine.[95] Arthur Norberg further argues that EMCC was pivotal in providing a coherent vision for the computing field as a whole.[96] The 1101, the EDSAC and the UNIVAC computing projects led to the adoption of three critical ideas that have become the backbone of modern programming: systems of notation to

---

[93] Ibid., 265.

[94] Ibid., 13.

[95] Beyer, "Grace Hopper and the Early History of Computer Programming," 72

[96] Norberg, *Computers and Commerce*, 9.

represent the manipulation of data, virtual addressing, and subroutines.

To understand these concepts, it is first necessary to explore early programming using machine language and the benefits of moving towards symbolic notation and manipulation, virtual addresses, and subroutines. The following example illustrates the complexity of summing two digits (Z = V+W) using the original schematics of the UNIVAC 1.[97]

The programmer begins by assigning three definite storage locations.[98] Hypothetically, the first could be for V, which is placed in the register location 001100. The second is for W at 002200. The final storage location is for the result, Z at 003300. According to EMCC documents contemporaneous to the time, the instruction for addition in the UNIVAC machine was located at 010100. Using this information and the hypothetical storage locations, the following string of digits would add the two numbers and place the results in the register location at 001100 (location V):

---

[97] This example was completed using Marvin L. Stein and David A. Pope, *Coding the Modern Digital Computer – With Special Reference to Univac Scientific Model 1103* (Minneapolis: University of Minnesota, 1960), 62 – 64; J.E. Sammet, *Programming Languages: History and Fundamentals,* (Englewood Cliffs: Prentice-Hall, 1969), 2; and Training Section of the Electronic Computing Department of Remington Rand, *Central Computer of the UNIVAC System, Manual of Operations,* (1954).

[98] In this case, we are talking about the programmer, not the coder who may be in the background inputting the program into the machine.

001100 || 010100 002200 003300

(V || + W Z)

While this is a trivial example, it demonstrates how complex it is to program directly into machine language.[99] Moreover, it almost completely ignores what else could be stored in the rest of the register – how do we know that there isn't already something stored at location 001100? The programmer would have to keep track not only of the valid storage areas in the register, but also of what register locations had already been used. Curiously though, for those of us looking back with hindsight, the idea of moving towards an abstract, symbolic notation, instead of coding directly in machine language, was not seen overwhelmingly as the best way of moving forward in programming. Martin Campbell Kelly illustrates in "Programming the EDSAC," that in the early years of computing, immediately following the introduction of the stored program concept, there were two fundamentally different approaches to programming. One segment of the fledgling community of programming theorists believed programmers should code directly in machine language, in octal (the base-8 number system) or binary code.[100] The other segment of this

---

[99] For further explanation of the symbols used in this dissertation, please see Appendix 3.

[100]In this system numerals can be made from binary numerals by grouping consecutive digits into lots of three. For example, 80 in binary is 1 010 000 — where the octal representation is 120. Further, we are talking about programmers writing the code in octal, versus a symbolic

burgeoning profession felt that there was a need for symbolic notation.[101]

Today, almost all programming is done using symbolic notation, where symbols (often English-like words) represent the underlying computational operations. This move towards symbolic notation was led by Maurice Wilkes and Admiral Grace Murray Hopper. Wilkes and Hopper were working in different countries on different projects. Hopper worked on the UNIVAC project in Philadelphia, while Wilkes was working on the EDSAC in Cambridge, England. Further, Wilkes and Hopper were working on different facets of the problem. Wilkes is best known for his development of the subroutine concept (although to do so he also created a system of symbolic notation), while Hopper is best known for her creation of a symbolic assembly program. These two developments are facets of the same problem. Both are attempts to ease the burden of programming by shifting the emphasis from the user to the machines. Together, Hopper and Wilkes' contributions guided the development of modern programming.[102]

---

programming language, however this written representation of the code would be given to a coder who would input it into the machine.

[101] Martin Campbell-Kelly, "Programming the EDSAC, 7.

[102] Kurt Beyer, "Grace Hopper and the Early History of Computer Programming,"; Campbell-Kelly, "Programming the EDSAC."

Sub-Routines and Virtual Addressing

Maurice Wilkes, during his term as the Director of the Cambridge University Mathematical Laboratory, built the first operational stored program computer in 1949.[103] The EDSAC was an EDVAC-type machine. The EDSAC processed serially, using mercury delay lines to implement the machine's memory. It was equipped with a binary machine language.[104] Wilkes adopted these features because he was influenced by the Moore School lectures.[105]

Wilkes and his team began work on a system for programming the EDSAC as soon as it became operational. Their most ground-breaking work was to produce a routine to input information into the machine, allowing the team to begin creating a library of subroutines. A subroutine is a short sequence of instructions written to accomplish a particular operation. As the computer moved out of the laboratory and into the hands of users, libraries of subroutines were stored on the machine. While the implementation of subroutines was not a new idea among computer designers (for example, Mauchly's 1947 paper, "Preparation of Problems for EDVAC type Machines" refers to

---

[103] Stern, *From ENIAC to UNIVAC*, 117n.

[104] Ibid., 117n, 123.

[105] Demonstrated by his attendance at this course. See Maurice Wilkes, *Memoirs of a Computer Pioneer* (Cambridge: MIT Press, 1985), 116.

subroutines as a way of building more complex problems) Wilkes' implementation was one of the first practical uses of the concept.

Implementing such libraries has one fundamental problem, now called the "relocation problem." At the time, Wilkes referred to it as the "adjusting process."[106] In these early years, programmers assigned definite storage locations for their data. If a programmer created a program that would add two numbers and stored their results in a specific storage location, every time the programmer used the program it would overwrite the results stored in the location specified by the programmer. Further, such a program could only use the numbers stored in the locations specified - it could not take two numbers as a parameter and then output the results. To solve these problems, Wilkes needed to formalize and implement two concepts: virtual addressing, which allowed the computer (instead of the programmer) to determine the best storage location for data and parameters, which allowed a program to accept parameters (data supplied by the user) instead of unique data as input. Both of these ideas resolved problems of complexity. Direct addressing was fraught with programmer error and allowing for libraries relieved programmers

[106] Campbell-Kelly, "Programming the EDSAC," 14 – 22.

from having to rewrite each small housekeeping task for each larger programming application.

Symbolic Notation and the Road to Automatic Coding Systems

Sort-Merge Generator

Francis Elizabeth Holberton began as one of the six female coders of the ENIAC project. Holberton left the Moore School with Eckert and Mauchly when they started the Eckert and Mauchly Computer Corporation, working directly with Mauchly.[107] One of the early projects Holberton worked on was the instruction code for the UNIVAC. This instruction code became known as the C-10 instruction set. Holberton developed the symbols used for representing operations and a three-digit designation for the memory location, written in decimal —not binary.[108]

Holberton is also well known for her creation of the Sort-Merge Generator for a magnetic tape computer in 1952.[109] The purpose of this generator was to effectively sort data, in the same manner as a punch card machine, as a method of utilizing the new electronic,

---

[107] Frances E. Holberton, OH 50. Oral history interview by James Baker Ross, 14 April 1983, Potomac, Maryland. Charles Babbage Institute, University of Minnesota, Minneapolis: 12.

[108] Beyer, "Grace Hopper and the Early History of Computer Programming," 180

[109] Frances E. Holberton, OH 50. Oral history, 43.

digital computer for business. The use of the UNIVAC for business was an important priority at EMCC. This is demonstrated in an early memo of the Future Design Group, which stated that the first project recommended by that committee for implementation was to create an instruction code suitable for "various large commercial firms."[110]

The Sort-Merge Generator was an important step in the history of software. Unlike the interpretive programs, the Sort-Merge Generator produced the program to conduct the sorting and merging from the specifications of the files, essentially, forcing the computer itself to write a program, not just translating completed programs. The Sort-Merge Generator defined the characteristics of the data to be sorted: the number of items, their length, and the different fields of data — to name a few. The operator entered a specific query and the generator would create another program which could instruct up to ten different tape drives to "copy" data from the different reels and then essentially "paste" it onto new reels. The process was iterative, continuing until only the requested data was on a final tape.[111]

---

[110]"Future Design Group" memo, August 15, 1950, Frances E. Holberton Papers, Box 13, Charles Babbage Institute, University of Minnesota, Minneapolis. Note, the Sort Merge Generator was reported in popular sources as being for the UNIVAC I, but it was not specified in either the memo or the interview with Holberton.

[111] Beyer, "Grace Hopper and the Early History of Computer Programming," 182 – 183.

Short Code

The first well-distributed tool that allowed programmers to use a more natural notation was Mauchly and William F. Schmitt's 1949 interpreter, Short Code. Schmitt remembers this as being a new project, although the essentials had been used previously in "Brief Code," which was written for the BINAC (the Binary Automatic Computer).[112] Short Code used a routine to interpret a program written in more natural notation, step by step, into machine language. At the completion of this process the entire translated assembly language program is executed.

Short Code translated its entire program into binary machine language, just like EDSAC, but it did not have arguments or parameters — increasing the difficulty of building a library of subroutines to be called at will. However, Short Code was another step towards automatic coding.

A0

It is Grace Murray Hopper's compiler, the A0, which traditionally has been considered the most influential tool in the quest for natural notation in programming. Currently, we define a compiler as a

---

[112] W.F. Schmitt, "UNIVAC Short Code," *IEEE Annals of the History of Computing* 10 no. 1 (1988): 11. Also note that this date is important because this is where the 1949 year arose – from the BINAC implementation. The successful UNIVAC implementation occurred after 1952, and therefore, after the EDSAC implementation.

program that takes a program written in a high-level language and turns it directly into another language — often first the assembly language which is then converted to machine language (binary code), which is then executed.[113] Hopper's early compiler worked in a different fashion. Instead, it used call-words to pull subroutines that were written in machine (usually binary) code out of a library. Following the call-words were the arguments, or parameters, to be entered into the subroutines.

The purpose of the A0 compiler was to execute mathematical programs, which would be used once, and produce an answer quickly, even for people who did not understand machine language.[114] However, the program itself took one minute longer than a hand coded program. This became significant when the same program had to be run multiple times. This is relevant when you consider the number of times the same equation needed to be performed to calculate the ballistic tables of the ENIAC or the number of times something like a sorting mechanism needed to be performed on a single data set.

The primary reason that Hopper's A0 was less efficient than later compilers is exactly what has changed in the current incarnation of

---

[113] Frank J. Galland, *Galland Dictionary of Computing* (New York: John Wiley & Sons, 1982), 46.

[114] Grace Hopper, "Keynote Address," in *HOPL-I: Proceedings of History of Programming Languages – I, 1981,* ed. Richard L. Wexelblat, (Blue Bell: Sperry Univac, 1981), 8-11.

compilers. Hopper's A0 did not translate line by line, rather, it compiled a series of programs. It then controlled the order in which these smaller programs ran and supplied them with their arguments, or parameters. As a result, the larger program did not know where the next subroutine began and used a lot of temporary storage during the transfer of control. Hopper's later compiler, the A2, addressed this problem, allowing the programmer to manually set entrance, exits and reference points.[115] Richard Ridgeway calculated that the A0 was eighteen times more efficient than traditional methods of programming when programming a basic equation. This efficiency was influential in demonstrating the importance of automatic coding systems for the growth of the software field.

With the introduction of the A2, Hopper had identified several major benefits of her work. Hopper used a more extensive, three address pseudo-code in the A2.[116] Hopper developed her understanding of implementing limited compatibility between the different models of the UNIVAC.

While Grace Hopper credited Betty Holberton's Sort-Merge Generator, A0, and Short Code at the 1978 History of Programming

---

[115] Beyer, "Grace Hopper and the Early History of Computer Programming," 199-205.

[116] Ibid., 205-209.

Languages Conference (HOPL), Beyer points out that in her earlier papers that influence is assigned more frequently to Wilkes and Mauchly, along with several much earlier figures, including Babbage, Leibniz and Aiken.[117] While it is always difficult to pinpoint influences after the fact, Wilkes' work was being carried out far removed from Hopper and EMCC. Further, the idea of a subroutine was entrenched in UNIVAC as early as 1949, and even as early as 1948 one of Holberton's flow charts refers to the use of a "Floating Decimal Subroutine for Addition, Multiplication, and Division."[118] Mauchly and Holberton were obviously immediate influences, engaging in dialogue about the programming of an existing machine. Tellingly, in an interview conducted in 1976, Hopper explained how even on the Mark I she was using libraries of small code segments.[119]

Complexity and Verification

Interestingly, there was a marked concern about verification of computer applications even in the earliest years of computing. Hopper, when working on Aiken's Mark I machine immediately found problems with accuracy of the programs she was creating, stating that

---

[117] Hopper, "Keynote Address," 7.

[118] Unpublished flowchart, "Floating Decimal Subroutine for Addition, Multiplication, Division," March 31, 1948, Frances E. Holberton Papers, Box 13, Charles Babbage Institute, University of Minnesota, Minneapolis.

[119]Grace Hopper, OH 81. Oral history interview by Christopher Evans, 1976, Science Museum, convenience copy at the Charles Babbage Institute, University of Minnesota, Minneapolis

"On the first early problems, when we were computing how much accuracy we were having, it was an appallingly rough computation," Hopper recalled. "We didn't know much about it yet."[120]

This is further supported when Frances E. Holberton anecdotally describes where the term "breakpoint" came from: in modern parlance, when debugging a program a programmer sets breakpoints, places where the instruction set stops, so that they confirm that results at that point are as expected. Holberton says that this term came from the ENIAC programmers setting breakpoints, by literally breaking the circuit, for the same purpose – to confirm that the results at that moment were as expected.[121] Throughout the Holberton interview there are references to the problem of accuracy, debugging, and verification in these early years of computing. [122]

"The third area was techniques of debugging, but really what I am referring to is testing the entire system. How do you prove that the [application] that you designed your program really works for "all conditions?"[123]

---

[120] Grace Hopper, in Kurt Beyer, *Grace Hopper and the Invention of the Information Age*, (Cambridge: MIT Press, 2009), 70.

[121] Holberton, Interview, CBI, p. 12

[122] Ibid.

[123] Danehower, Interview, CBI 200, p. 134

Further, there was a divide about how to address errors in the computing community. One group wanted to address errors with equipment that would detect errors and stop the program, while others in the community wanted to prevent errors from occurring at all. This debate over the priority of error prevention versus error handling in computer applications has raged from the inception of computing through to current articles on verification.[124]

Complexity was the driving force behind the creation of symbolic programming. Even in the earliest years of programming it is evident that programming was difficult, I have tried to demonstrate the complexity of software in the chapter. Nathan Ensmenger further reiterates this in his book, *The Computer Boys Take Over*, arguing that "programming…was an inchoate discipline, a jumble of skills and techniques.[125]

As Douglas T. Ross said in an oral history interview archived at the Charles Babbage Institute,

> "You hear about software, especially coming from an engineering place, that you can always fix it up by changing the software. That's a lot easier than the hardware. Well, actually, there's nothing more

---

[124] J Forrester, "Digital Computers: Present and Future Trends," in *Joint AIEE-IRE Computer Conference: Review of Electronic Digital Computers, December 10-12, 1951,* (New York: American Institute of Electrical Engineers, 1952).

[125] Ensmenger, *The Computer Boys Take Over*, 58.

> hard than software -- and I don't mean hard to write. It's actually brittle." I gave them this example of one bit making the whole thing shatter and be no good at all. There is nothing else that is that brittle. I think of it like a crystal structure splitting. That was my classic example."

This is why Hopper's work is so crucial. A co-worker of Grace Hopper once said about her, "She simplified this tool in effect or the utilization of it so the average person could use that tool."[126] This need to simplify a complex problem would become a theme throughout the history of software.

[126] Beyer, "Grace Hopper and the Early History of Computer Programming," 72

Chapter 3: Automatic Coding Systems

Advent of Automatic Coding Systems

The advent of subroutines, symbolic notation, and virtual addressing in the late 1940s set the stage for automatic coding systems. As the *4th Edition of the Dictionary of Computing* tells us, the phrase "automatic coding" is obsolete. In the early 1950s, though, automatic coding signified the difference between hand coding and the use of what was at the time, primitive, high-level programming languages. A high level language refers to programming languages where instructions or statements correspond to machine language instructions.[127] The reason that the phrase "automatic coding" became obsolete is because it is practically unheard of to hand code directly into binary or machine language anymore. While automatic coding systems were intended originally to replace the binary notation with a symbolic notation that would ease the burden on programmers, it quickly became obvious that they could also automate the routine clerical work of the programmer and overcome hardware shortcomings.

Before diving into the details, I am going to provide a brief overview of how automatic coding systems worked on a practical level.

---

[127] I.C. Pyle and Valerie Illingworth, eds., *A Dictionary of Computing, 4th Ed* (New York: Oxford University Press, 1997)

After designing the program, the programmer would punch the code (in a high level language like FORTRAN) on to punch cards (this could be any input/output medium, including magnetic tape), often using a keypunch machine. This is the source code. In large institutions the programmer would then deposit the deck of cards with the program represented in a high level language at the computer center or similar division. The operator would run the deck of cards through the computer, using the interpreter or compiler. This produced a second deck of cards; this deck was in binary code. It is this second deck of cards that has the executable (binary) code and the operator would then feed it back into the computer to run it as a functional program, performing the operations the programmer originally designed. While we often talk about "punched cards", the fact that the translated binary program (the object code) is a separate, physical embodiment (either in a card deck, magnetic tape, or another storage medium) is often overlooked.

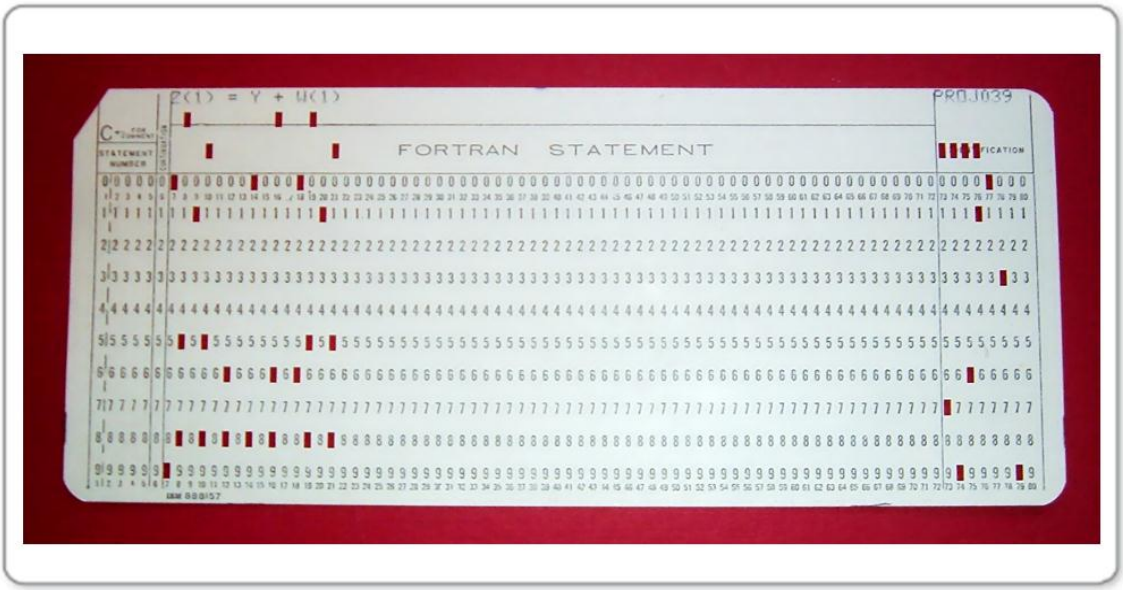


Figure 5: FORTRAN Statement on Punched Card
Image provided by Arnold Reinhold, through Wikimedia Commons License

Clerical functions were commonly subsumed by the automatic coding systems. These systems often had libraries of subroutines. These libraries often included input/output and memory management routines – routines needed by most programs regardless of their larger function. Automatic coding systems also implemented virtual addressing - storing instructions and data in appropriate locations in the computer's register using variables instead of a programmer-supplied binary address. Prior to the advent of automatic coding systems, programmers had to define (address) where each instruction and each variable (piece of data) would be stored. This address was a string of zeros and ones that specified the location in the register.

Automatic coding systems also alleviated existing machine shortcomings, by providing functions that the hardware alone could not incorporate. One specific example of this use of coding systems was tools that would facilitate floating point arithmetic.[128] Floating point arithmetic refers to the storing and calculating of numbers where the decimal points do not line up as in integers. Instead, the significant digits are stored as a unit (the mantissa) and the location of the decimal point is stored in a separate unit (the exponent). The purpose of floating point methods is to quickly calculate a large range of numbers. It was difficult to implement this in hardware because of the space requirements necessary, so it was instead implemented in the software of the early automatic coding systems.

These uses of automatic coding systems bisect the problem of complexity. Automating the routine clerical work simplified programming, because programmers no longer had to re-write routine functions (like input/output functions) that were necessary for every significant program. Not only did this ease the time burden of programming, it also prevented the normal errors (commonly called bugs) that cropped up in these functions, relieving the need to debug

---

[128] John Backus, "The History of FORTRAN I, II and III," in *HOPL-I: Proceedings of History of Programming Languages – I, 1981*, ed. Richard L. Wexelblad, (Blue Bell: Sperry Univac, 1981), 25.

different implementations of the same functions. Subsuming the repetitive, highly mathematical work of addressing (storing instructions and data in appropriate locations in the computer's memory) prevented transcription and other human errors common in the addressing. Hopper talks specifically about transcription errors and addition errors in these routine tasks as the inspiration for her early automatic coding systems, stating she "sure found out fast that programmers cannot copy things correctly…Programmers could not add. There sat that beautiful big machine whose sole job was to copy things and do addition. Why not make the computer do it?"[129]

That an automated tool would take over the mundane, repetitive tasks of programmers was not accepted without debate. A subset of the industry preferred to code directly in the language of the machine, seeing automatic coding systems as inefficient and error-ridden.[130] Maurice Wilkes, quoting John W. Carr, described this group of the computing community as "primitives." This blow was softened by naming proponents of the system, himself included, as "space cadets." But by the late 1950s the use of automatic systems was seen as the future of the computing industry and the "space cadets" had clearly

---

[129] Captain Grace Hopper interview by Angeline Pantages, December 1980, Naval Data Automation Command, Maryland (Computer History Museum, CHM Reference Number X5142.2009).

[130] Campbell-Kelly, "Programming the EDSAC," 7

won.[131] The complexity and the scope of software projects were growing significantly, both because of the rapid adoption of computing technology, but also because of the new industries that were adopting computers – finance, insurance, and travel.

Despite consensus about the future of automatic coding systems throughout these early years, there was still a philosophical division in the programming industry. Some programmers saw automatic coding systems as a tool for novice users to program a complex machine-the computer.[132] Others saw automatic coding systems as tools to facilitate the work of an educated programmer.[133] The two most popular automatic coding systems embody the differences of these early computing philosophies. FORTRAN can be seen as a system that could facilitate the work of a programmer, while COBOL was intended to be used by more novice business users.

FORTRAN

> I don't know what the language of the year 2000 will look like but I know it will be called FORTRAN.
>
> Tony Hoare[134]

---

[131] Maurice V. Wilkes, "Computers Then and Now," *Journal of the ACM* 15 no. 1 (1968): 1-7.

[132] Backus, "The History of FORTRAN I, II and III," 25

[133] Eldridge S. Adams, Jr., "Simple Automatic Coding Systems," *Communications of the ACM*_ 1 no. 7 (1958): 5-9.

[134] J.A.N. Lee, "Pioneer Day," *IEEE Annals of the History of Computing* 6 no. 1 (1984): 13 (as a saying from a card distributed at the Pioneer Day event).

John Backus began work on FORTRAN (FORmula TRANslator) in 1954. However, his interest in automatic coding systems began earlier. In 1953 he developed a simple alternative to machine language coding which he named Speedcode. Speedcode originated as a method to facilitate Backus' own work, which was scientific in nature.[135] This influenced the orientation of his early language design, supporting scientific uses, as opposed to business uses, of the computer. Backus was further influenced by the existing automatic coding systems. He saw those systems as inefficient and inaccurate. This inaccuracy of the underlying programming made it difficult to verify the programs built on these programs. Moreover, during this time the computing costs, represented by the running times, of programs were not insignificant and inefficient programs were expensive.[136] However, without automatic coding systems the costs of programming were often equal to the costs of the computer itself.[137]

IBM recognized this problem early. Programming costs, both in development and in debugging, detracted from a computer's projected cost benefits. In early studies, IBM found that when it was online the

---

[135] Backus joined IBM in 1950. His first major project was to write a program to calculate positions of the Moon. John Backus, interview with Grady Booch, September 5, 2006, page 8, transcript (Computer History Museum, Ref. No X3715.2007).

[136] Backus, "The History of FORTRAN I, II, and III," 27.

[137] While I explored John Grisham's corpus of work to illustrate this, he did not address these topics.

computer was spending between one-quarter and one-half of its time de-bugging existing programs.[138]

To illustrate one significant example of the problems of the then-existing automatic coding systems, Backus relates that one of the most common uses of automatic coding systems had been to solve the problem of floating point calculations. However, many vocational programmers blamed the speed of their automatic coding systems on the floating point arithmetic, when really they were using this as an excuse to lower the expectations for the performance of the systems. They used this excuse as a way to obfuscate the inefficiency of the techniques they used for their other tasks - like the simulated indexing needed for a virtual address.[139] When the 704 was built with efficient floating point arithmetic hardware this exposed the inefficient techniques used to handle other tasks.[140]

While the initial push for FORTRAN came from Backus, he worked closely with Irving Ziller, Harlan Herrick, Robert Nelson, Roy Nutt, and other members of the "Programming Research Group." The group emphasized the efficiency and accuracy of the object program,

---

[138] Ibid., 26. Backus argues that the system often slowed the machine "by a factor of five or ten."

[139] Ibid.,

[140] Ibid., 28.

the program that would be compiled and run by the computer, over the more abstract goal of elegant language design, which would be written by human programmers. Backus discussed this choice at the FORTRAN session at the first History of Programming Languages Conference, arguing that they never saw the language design as a difficult problem. Instead, they saw designing the language as a preliminary step to the complicated work of designing a compiler that could produce efficient, accurate object programs.[141] The efficiency and accuracy of the object programs was what would delineate FORTRAN from the existing automatic coding systems.

Neither Backus nor IBM ever expected FORTRAN to become the primary scientific programming language for over twenty years. As seen in the preliminary report on FORTRAN, the purpose of FORTRAN was to improve programming on the IBM 704 and to pressure other manufacturers to provide similar systems that would further reduce the cost of programming.[142] In the preliminary report Backus defined the needs of the language to include variables, function names, recursively defined expressions, arithmetic formulas and Do formulas (where Do formulas were specifying both the first and last statements as well as a third statement to which control could pass). These

[141] Ibid., 30.

[142] Ibid.

components remain standard features of programming languages to this day.

FORTRAN was wildly successful. Programming specialists have remarked that FORTRAN "established the feasibility of using high level languages."[143] FORTRAN's compiler was persuasive because it optimized the code. The assembly language produced by the FORTRAN compiler had efficiency comparable to code produced by a vocational programmer. Further, when we compare a small algorithm written in Hopper's A2 with the same algorithm in FORTRAN, the difference is obvious. FORTRAN implements the code in several easily understandable sentences, in comparison to the lines of characters required by A2.[144]

FORTRAN has remained the leading scientific programming language since the 1950s. It has remained at the forefront for a number of reasons. FORTRAN has algebraic expressions and a rich mathematical language. It standardized early, beginning in 1962. This allowed FORTRAN to develop a significant user base, which could rely on the same techniques to work on different machines. Martin Greenfield argued that FORTRAN's standardization unified the voice of

---

[143] Ian D Chivers, Malcolm W Clark, "History and future of FORTRAN", *Data Processing*, 27 no.1 (1985): 39 - 41

[144] See Appendix 1 for details

the programming industry. Because the language was standard over different computer systems vocational programmer's skills became transferable.[145] Moreover, while FORTRAN has undergone significant updates, FORTRAN's compatibility was preserved. This benefit of compatibility can be seen throughout the history of programming, in the compatibility of C++ with C because of the broad user base, but also because of the problem of legacy code. Legacy code is the code for applications written years earlier that still need to be maintained and extended. FORTRAN was used in systems that have a long life span. For example, both Livermore National Laboratory and Westinghouse-Bettis nuclear power reactor development laboratories were early adopters of FORTRAN and still use the language in current applications.[146]

---

[145] Martin N. Greenfield, "History of FORTRAN Standardization" in *AFIPS Conference Proceedings, 1982 National Computer Conference, June 7-10 1982,* ed. Howard Lee Morgan, (Arlington: AFIPS Press, 1982.)

[146] HS Bright, *Computers and Automation* 20 no. 11 (1971): 17-18; J.A.N. Lee and H.S. Tropp, eds., "FORTRAN's Twenty-Fifth Anniversary," special issue, *IEEE Annals of the History of Computing* 6 no. 1 (1984); Robert A. Hughes, "Early FORTRAN at Livermore," in "FORTRAN's Twenty-Fifth Anniversary," ed. J.A.N. Lee and H.S. Tropp, *IEEE Annals of the History of Computing* 6 no. 1 (1984): 30-31.

COBOL

Figure 6: COBOL Cartoon, Toonlet.com, 2008

> COBOL is a very bad language, but all the others are so much worse.
>
> Robert Glass[147]

COBOL (COmmon Business-Oriented Language) was created with a different philosophy and in a different atmosphere from FORTRAN. COBOL was the first language to be created by committee. It was designed to fulfill a perceived need for a business-oriented

[147] Robert L. Glass, "Cobol - A Contradiction and an Enigma", *Communications of the ACM* 40 no. 9 (1997): 11-13

language. The goal for the committee was to create a language simple enough to be used by novice computer users and under supervision by management with no computing experience. In April of 1959 representatives of users, academia, and manufacturers attended a meeting at the University of Pennsylvania's Computing Center to plan a formal meeting on common business languages.[148] It was in this early, informal meeting that the committee decided to approach the Department of Defense about sponsoring a project to create a common business programming language, and the committee structure was formed.[149]

The original committee for the business-oriented language was broken up into three sub-committees to address the short-, intermediate- and long- term goals of the project. However, only the short-term and intermediate-term committees were ever convened. The COBOL executive committee, named CODASYL (Conference on Data Systems Languages), was responsible for defining the elements of the language, while computer manufacturers would instantiate the COBOL compilers for their respective machines. Like the FORTRAN

---

[148] Jean Sammet, "The Early History of COBOL," in *HOPL-I: Proceedings of History of Programming Languages – I, 1981*, ed. Richard L. Wexelblat (Blue Bell: Sperry Univac, 1981), 200.

[149] This early sponsorship by the DOD is the foundation for the popular understanding of COBOL as the result of a DOD project. However, as Sammet explains in her paper, "The Early History of COBOL," the Department of Defense's involvement in the project was secondary to the involvement of users, academia and manufacturers.

group, CODASYL believed that the language was a preliminary or interim language to facilitate the business use of computing, but would not be a lasting standard.[150]

Unlike the FORTRAN group, COBOL was originally developed as a composite language. The CODASYL committee used existing languages, (FLOW-MATIC, AIMACO, and COMTRAN) to guide their design of COBOL.[151] This composite approach was a result of the committee's belief that the resulting COBOL language would be an interim measure, so for expediency, instead of developing new concepts, the committee intended to combine the best elements of existing programming languages. The use of a verb to start every command, the use of the GO TO statement, and over 400 of the keywords chosen were adopted from existing languages.[152]

COBOL was designed around three central characteristics – natural notation, ease of use, and versatility across different systems. CODASYL's intention with this design philosophy was to create a language that could be used by a novice, vocational programmer and

---

[150] Sammet, "The Early History of COBOL," 200.

[151] Please note, there is some contention that Honeywell's FACT was integral to the development of COBOL. This stems from an acknowledgment in the COBOL documentation, but Jean Sammet refutes this influence, illustrating with dates that the COBOL language description was complete prior to the committee being introduced to the FACT documentation.

[152] Sammet, "The Early History of COBOL," 221-235.

was easily understandable by traditional managers. The ease of use and understandability of COBOL is evident in the following example:

```
DISPLAY "Enter number: "
ACCEPT Num1
DISPLAY "Enter number:"
ACCEPT Num2.
SUBTRACT Num1 FROM Num2 GIVING Result.
DISPLAY "Result is = ", Result.
STOP RUN.
```

Figure 5: Example of a COBOL program
The program allows the user to input two numbers & performs the subtraction. It then displays the result.

Three languages were chosen to supply the language elements that made COBOL so user friendly. FLOW-MATIC (originally named B-0) was Grace Hopper's companion to the A0 and A2 languages developed for the UNIVAC 1 at Remington. FLOW-MATIC had an English-like language structure. COBOL incorporated the verb choices and elements of the input/output systems found in FLOW-MATIC.[153] AIMACO (AIr MAterial COmmand compiler) was defined by a committee composed of industry representatives and chaired by a representative of the AMC - the Air Material Command arm of the US Air Force. The CODASYL committee replicated AIMACO's file design.[154] COMTRAN (COMmercial TRANslator) was a 1957 programming language developed at IBM by Bob Bemer. Like the parallel between

[153] Ibid., 204-205.

[154] Ibid., 221-235.

B-0 and A-2 COMTRAN was intended to be the business-oriented equivalent of FORTRAN. The COBOL committee was influenced by COMTRAN in assigning names to routines and then used that name to specify the destination of "go to" statements – as opposed to the line number. The Picture Clause (an element determines characteristics of a datum - the format, type and size for example) was also first implemented in COMTRAN before being incorporated in COBOL.[155]

COBOL is, undeniably, still one of the most frequently used languages. *Computerworld* conducted a survey of 352 IT managers in early 2006 and 62% of the respondents were actively using COBOL.[156] COBOL is the second "peak" in Martin Campbell-Kelly's "twin peaks of FORTRAN and COBOL [which account] for two thirds of the applications programming activity of the 1960s and 1970s."[157] One well-known ramification of this was the "second gold rush" for vocational COBOL programmers during the lead up to the turn of the Millennium,[158] which illustrated how many major computing systems were relying on

---

[155]Ibid.

[156]G. Anthes, "COBOL Coders: Going, Going, Gone?" *Computerworld*, October 09, 2006, http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=266228

[157] Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog*, 36.

[158] Because of the Y2K bug, many of the legacy applications written in COBOL needed to have dates altered from the 2 digit format to the 4 digit format (from 07 to 2007, for example).

legacy software of the 1960s and 1970s.[159] Jim Cortada estimated that between sixty-five percent and eighty-five percent of all business applications were written in COBOL.[160]

There are several reasons for COBOL's long-lasting popularity. First the active involvement of mainframe manufacturers in the definition of the language ensured the creation of compilers for a wide variety of systems. This made the language easily accessible and standardized across different platforms. Vocational programmers and programming specialists alike knew they would both have access to the language and that it would be the same despite moving from one computer system to another. A second reason for COBOL's popularity was that the Department of Defense recommended the use of COBOL in projects they sponsored, stating that "COBOL would be the preferred language for problems classed as business data processing."[161] This resulted in a large number of applications written in COBOL and a significant number of both vocational programmers and programming specialists using the language on DoD sponsored projects. COBOL is still in use in a large number of legacy systems —

---

[159] M. Brandel, "The Top 10 Dead (or Dying) Computer Skills," *Computerworld*, May 24, 2007, http://www.computerworld.com/action/article.do?command=printArticleBasic&articleId=9020942.

[160] Cortada, *Historical Dictionary of Data Processing Technology,* 168-270.

[161] Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog*, 36.

systems that have needed to be maintained for upwards of forty years. Finally, the simplicity and ease of use of COBOL's syntax has contributed to vocational programmers preferring it over other languages with a more complicated syntax.

FORTRAN and COBOL solved the complexity of hand coding directly in machine language by delivering libraries of subroutines, handling the addressing and other routine clerical tasks required for programming and providing an English-like syntax. However, automatic coding systems like FORTRAN and COBOL also add a layer of complexity, by keeping the programmer one step removed from the machine language. This also results in problems of verification, because while the programmer can verify that the code in their program is correct, the underlying machine code is, essentially, black-boxed.

The wide acceptance of FORTRAN and COBOL by the vocational software community has made them synonymous with the acceptance of automatic coding systems. However, other languages were being developed that would later prove influential in the debate over structured programming and the later shift to object oriented programming. Two specific languages, ALGOL (ALGOrithmic Language) and LISP (LISt Processing), both played significant roles in

the later changes in the history of software. However, both ALGOL and LISP are programming languages developed exclusively for uses internal to the community of programming specialists. There was no sense that casual computer users would use either ALGOL or LISP. In this sense, ALGOL and LISP increased the complexity of programming.

ALGOL

> Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.
>
> C.A.R. Hoare[162]

> ALGOL 68 is like an Aston Martin. An impressive car but not just anyone can drive it.
>
> Unknown[163]

ALGOL began as an attempt at international co-operation to resolve problems of communication in computing. Originally titled the International Algebraic Language, ALGOL was intended, like FORTRAN, as a language primarily designed for the scientific community. In the late 1950s, the community of programming specialists perceived a need for a universal algebraic notation for programming languages. The 1955 International Symposium on Electronic Digital Computers and Information Processing in Darmstadt is credited as the epicenter of this discourse. The idea of having a universal programming

---

[162] C. A. R. Hoare, *Hints on Programming Language Design*, Stanford University: Stanford, 1973

[163] "That was Funny," Accessed on July 27, 2010, http://www.thatwasfunny.com/programming-languages-are-like-cars/1509

language for scientific use took root, and in 1957 the ACM and the GAMM ("Gesellschaft für angewandte Mathematik und Mechanik") agreed to work together on this task.[164]

Like COBOL, ALGOL attempted to compile the best features of existing language, without including the bloated and unnecessary features that were becoming common in automatic coding systems. ALGOL used many of the elegant features of FORTRAN and FLOW-MATIC, separating function and procedure, and the same basic logical features of those languages - types, conditionals, loops, and switches.[165]

ALGOL was not a popular language in commercial programming. However, it was very popular among programming specialists. ALGOL was used almost universally as a language to frame questions about programming theory, appearing prevalently in broader computing journals and more narrowly focused journals dedicated to programming theory. ALGOL was seen as a language that was consistent across both continents and machine platforms. ALGOL was also influential in that it inspired what we now call algebraic

---

[164] D. Nofre, "Unraveling ALGOL: US, Europe, and the Creation of a Programming Language," *IEEE Annals of the History of Computing*, 32 no. 2 (2010): 58-68.

[165] Alan J. Perlis, "The American Side of the Development of ALGOL," in *HOPL-I: Proceedings of History of Programming Languages – I, 1981*, ed. Richard L. Wexelblat, (Blue Bell: Sperry Univac, 1981), 90.

programming languages, a type or style of programming languages. Pascal, MAD (the Michigan Algorithm Decoder), and JOVIAL (Jules Own Version of the International Algorithmic Language) are well-known languages derived from ALGOL.[166]

What set ALGOL apart from other automatic programming languages of the time was because ALGOL was described in Backus-Naur Notation (Backus-Naur Form, or BNF). The importance of this style of notation cannot be overestimated. BNF is extremely powerful. It describes the grammatical structure of a programming language without room for uncertainty, allowing the language to be implemented without machine dependency, resulting in compatibility across different manufacturers and models. Since the introduction of ALGOL, BNF or Extended BNF (EBNF, which addresses recursion) has been used to describe all programming languages.

BNF describes the grammar of a programming language, so for example when describing a programming language in BNF notation the syntax of the language is declared like this:

<syntax> ::= <rule> | <rule> <syntax>

---

[166] Ibid.

To clarify with a real example, in SQL (the popular database programming language) one element of the language is Alter Table. This command allows the user to alter the number or the name of columns in the database table. For example, if you had a database to track students, the table columns may be "student name", "student id" and "student address." If in the future, the table needed to be changed to include students' phone numbers, the Alter Table statement could be executed on the database, changing the table to include a new, blank, column named "student phone number."

When SQL is defined using Backus-Naur Form notation the Alter Table command is defined like this:

<alter table statement>::= ALTER TABLE <table name><alter table action>

The alter table action is then defined to be adding a column, altering a column, dropping a column, adding, or dropping a constraint from the table and is described like this: (Note that the symbol ::= is an assignment operator)

<alter table action> ::=
<add column definition>
| <alter column definition>
| <drop column definition>
| <add table constraint definition>

| <drop table constraint definition>

Backus Naur Notation allowed for many different languages to be defined in the same way, creating consistency in language definitions and making it easier to both define new languages and to learn new programming languages. Noticeably, throughout this dissertation I have used elements of BNF notation and standardized pseudo-code to illustrate my examples. Anecdotally, even with only a Bachelor's level of understanding of programming, it has become second nature for me to rely on BNF notation to clearly describe a programming problem unencumbered by the specifics of a single programming language.

The purpose of ALGOL was to simplify programming by standardizing it globally. It was thought that this would improve the free exchange of ideas. ALGOL was also intended to redress similar problems that FORTRAN and COBOL resolved. It provided symbolic notation, as opposed to binary. ALGOL included subroutines that would subsume the clerical responsibilities of the programmer and handled addressing. Through the use of BNF, it was intended that ALGOL would be implemented consistently across the different computer manufacturers and models. While ALGOL itself never standardized programming globally, BNF, which came out of the work on ALGOL, did create a worldwide *de facto* standard.

ALGOL, or more specifically, the introduction of a standardized notation with which to define programming languages, decreased the complexity of programming. As a result of BNF, when learning a new syntax, vocational programmers knew exactly what each keyword would do when used. Vocational programmers could easily comprehend the arguments (inputs) that new functions required. However, BNF also simplified the act of defining a new programming language. This inadvertently helped to create a swarm of new programming languages and increasing the complexity of programming by providing a multitude of new options when choosing a language. Soon after BNF notation was introduced PL/I, SNOBOL, APL, CPL, and BASIC were all released and defined using BNF notation. Verification became less attainable as the languages became more high-level, often condensing numerous assembly language routines into a single high level function. However, Peter Naur's contemporaneous work with his contributions to ALGOL and BNF notation was on formal mathematical verification. Naur's work on verification wasn't a sophisticated proof system – but it was heavily based on mathematics.[167] This highlights the relationship between programming theory and verification.

---

[167] MacKenzie, Mechanizing Proof, 49

Programming theory has a debt to ALGOL. BNF notation, typed languages and other features of ALGOL have become standard elements of modern programming. However, ALGOL never became a popular programming language in the United States. In the early 1960s the ACM, IBM, and IBM's user group SHARE withdrew their support for the language.[168] Despite ALGOL's limited role in commercial applications, it is integral to much of the debate and discussion in the 1960s and 1970s that eventually culminate in the paradigm shift towards object oriented programming.

[168] D. Nofre, "Unraveling Algol," 60.

LISP

Figure 7: LISP Cartoon, Kylie Miller and John Zakour, 2006

> Overall, the evolution of LISP has been guided more by institutional rivalry, one-upsmanship, and the glee born of technical cleverness that is characteristic of the "hacker culture" than by sober assessments of technical requirements. Nevertheless this process has eventually produced both an industrial-strength programming language…and a technically pure dialect.
>
> Guy Steele and Richard Gabriel[169]

John McCarthy developed the concept of LISP. Like ALGOL and FORTRAN, LISP was also created to provide a practical mathematical notation for programming specialists. McCarthy first implemented an abbreviated version of these ideas in the FORTRAN List Processing

[169]Guy L. Steele, Jr. and Richard P Gabriel, "The Evolution of LISP," in Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. eds., *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993* (New York: ACM, 1994).

Language in the mid to late 1950s. Between 1958 and 1962 McCarthy worked on implementing the LISP programming language.[170] LISP was different from other automatic coding systems because it performed computations with symbolic expressions – not numbers. McCarthy achieved this by representing the expressions and data using a list structure. LISP source code is itself, a list and therefore, can be manipulated in the same way as any other data. This interchangeability of code and data allows an application written in LISP to modify the language itself – meaning that a LISP application can change a function of the programming language or write a new function for the language with ease. In traditional languages, to manipulate a function or create a new function within an application would require complex parsing or even manipulation of the assembly language. LISP was also one of the first languages to implement recursion and garbage collection.[171] It is probably best known for its elegance and simplicity. LISP users are often very loyal to the language.

McCarthy's intent with LISP was to find an algebraic list processing language for use on the IBM 704 to research Artificial

---

[170] John McCarthy, "History of LISP," in *HOPL-I: Proceedings of History of Programming Languages – I, 1981*, ed. Richard L. Wexelblat, (Blue Bell: Sperry Univac, 1981), 173.

[171] Ibid.

Intelligence. With this pedigree, it is unsurprising that LISP became the natural choice of the artificial intelligence community.[172] McCarthy's list structure was particularly appropriate for AI programming because the internal list representation provides a less complex notation for logical deduction, algebraic simplification, differentiation and integration. Moreover, McCarthy was part of the group of programming specialists that were most interested in mathematizing programming theory, both as a way of creating elegance (or, in the language I am using in this dissertation, simplifying programming) and as a step towards formal, mathematical verification.[173]

LISP has specific and significant impacts on the process of verification because it can be used to write a new programming language. Because the new programming language is already written in a high level language itself it is one more step removed from the machine language of the computer. This introduces anonymous subroutines — LISP routines (made up of numerous assembly language commands) that are running in the background of the new programming languages' functions and routines. This increases the complexity of the program, making it harder to verify mathematically

[172] Ibid., 177-179.

[173] MacKenzie, Mechanizing Proof, 48

or automatically, while decreasing the complexity of programming, allowing for larger scale software.

Outside of the elite AI community, LISP was not frequently used, but praised frequently. The elegance of LISP has been very influential. Moreover, while McCarthy's primary interest was in developing a logical formalism to express every day reasoning, his secondary interest was in verification. In 1962, McCarthy had already created an application to verify the correctness of a compiler hand in hand with his work on LISP.[174] LISP inspired Alan Kay's development of Smalltalk, which was a significant part of the evolution of object oriented programming.

ALGOL & LISP: the theoretical equivalent of FORTRAN & COBOL?

ALGOL and LISP were two languages that were influential in the future of programming theory, despite the limited popularity they had in commercial programming. In 1981, Alan Perlis commented on the limited popularity of ALGOL, arguing that the community of programming specialists in the United States was ambivalent towards

[174] Ibid.

ALGOL and thought of ALGOL as an elegant language but useful only for publishing technical issues in the literature.[175]

A brief literature review, graphically presented in figure 8, demonstrates how important ALGOL and LISP were to programming theory, despite their limited use in a commercial setting. During the time period I examined, more articles were published about ALGOL than on FORTRAN and COBOL combined. This is not unexpected since the ACM was, in the beginning, one of the supporters of ALGOL, but by 1965 they had withdrawn their support. [176] Further, almost as many articles were published on the boutique language, LISP, as on COBOL. The "in the trenches" programmers were contributing little to the theoretical literature, but many articles published by the ACM on tricky or interesting software implementations were written by vocational programmers.

I argue that the high number of articles written about LISP and ALGOL, despite their limited use (estimated to be used in less than 10% of software applications) illustrates one divide in the broader software community. Programming theorists were out of touch with the needs of the vocational programmers, but alternately, vocational

---

[175] Perlis, "The American Side of the Development of ALGOL," 87.

[176] Ibid.,76.

programmers were interested in the day to day implementation of software and were neglecting to ask themselves what comes next. There was little debate between these groups until their interests collide in the late 1960s and early 1970s.

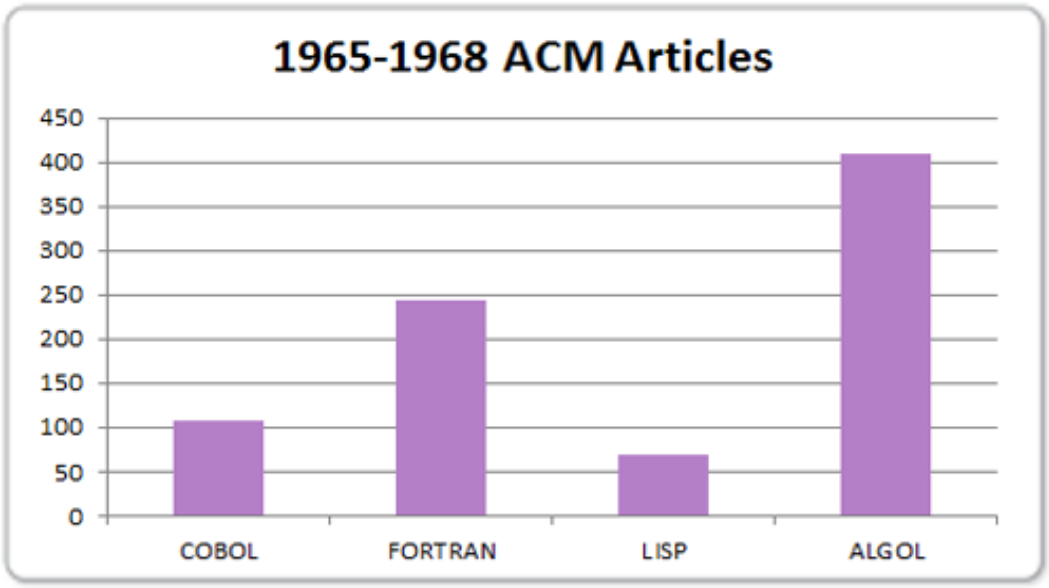


Figure 8: Number of ACM Articles on COBOL, FORTRAN, LISP and ALGOL Between 1965 and 1968

While this vocational programmer/programming specialist divide has been explored in Nathan Ensmenger's work, *The Computer Boys Take Over* and in Thomas Haigh's dissertation, "Technology, Information and Power", I suggest there is another, more subtle divide at work during this time period. This period illustrates early elements of a later schism between the programming theorists. In their desire to overcome problems of complexity and to attain the goal of

verification, there are those that intend to mathematize programming theory and those that prefer a more pragmatic approach. This schism within the programming theory community will become evident in chapter four.

Complexity, Verification, and Automatic Coding Systems

By the early 1960s almost all programs were coded in programming languages. While this is usually presented in the traditional narrative as technological determinism — that programming languages were simply better than the alternative — this was not the case. Jean Sammet argued that in the early years of programming theory high level programming languages were a "unique and generally uncomfortable concept." Instead of the unconditional acceptance of the better alternative, in the literature we see J.W. Carr's "space cadets" lobby and persuade their "primitive" adversaries with code optimizing compilers, elegant notation and increased efficiency. In a true example of history being written by the victors, the later luminaries of programming theory were the champions of the movement towards an automatic coding system. Grace Hopper, John Backus, John McCarthy and Peter Naur - all are names we see repeatedly and consistently throughout the early years of

programming theory and all were actively promoting the use of automatic coding systems.[177]

Moreover, we see John McCarthy at the forefront of the work being done on verification. As one of the pioneers of AI, MCCarthy worked on the mathematization of computer science, as a goal towards verifying that procedures solved the supplied problem set. Peter Naur, Robert Floyd, and Tony Hoare were all pursuing this goal of mathematical verification in the 1960s, contemporaneous to the advent of automatic coding systems.[178] This illustrates the significant role that verification played, even during the earliest years of programming.

Automatic coding systems simplified the way programming specialists and vocational programmers, alike, write software. Throughout this chapter we see that the widespread adoption of symbolic notation, libraries of subroutines and virtual addressing improved the efficiency of programming and accuracy of programs. However, these systems also decreased barriers to entry, flooding the field with new, vocational programmers with different backgrounds.

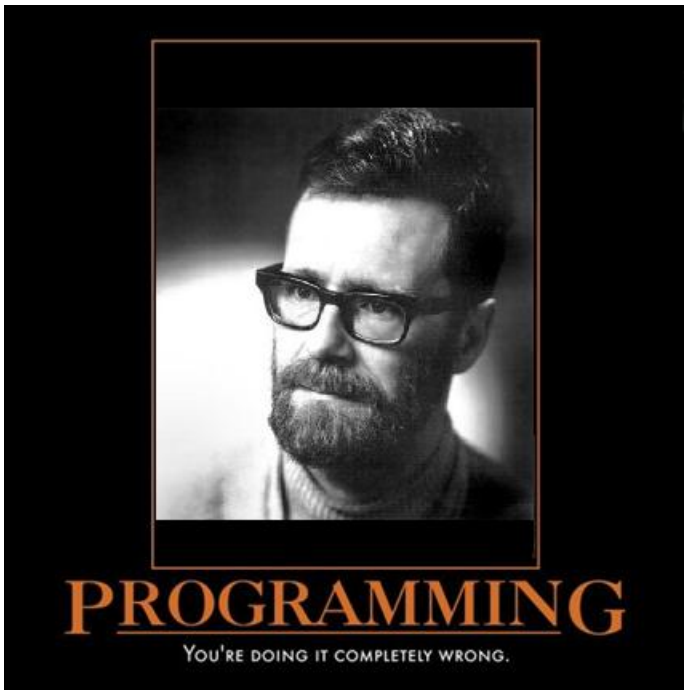
---

[177] One issue that I don't address in this dissertation is why there is such a push for universal programming languages like ALGOL. Priestley addresses this in his dissertation, arguing that it is a push for reusability and simplicity. Peter Mark Priestley, "Logic and the Development of Programming Languages, 1930 – 1975", 130

[178] MacKenzie, *Mechanizing Proof*, 49

They increased the scope of software applications, by increasing the efficiency of the programming process. This led to demand for more vocational programmers, further expanding the size of the field. Automatic coding systems both increased accuracy and created more difficulties in the verification of programs. All of this resulted in significant discourse throughout the community of programming specialists on the topics of both complexity and verification. Attempted solutions to the problems posed by increasing complexity of software and a decreasing ability to verify the accuracy of the programs became common.

Chapter 4: Structured Programming



> The central problem of Computer Science seems to be: how can we design highly sophisticated systems in such a controlled manner that otherwise unmanageable complexity is carefully avoided?
>
> E.W. Dijkstra[179]

Discontent

Structured programming is a phenomenon of the late 1960s and 1970s. Edsger W. Dijkstra, a charismatic, outspoken, and controversial leader in the community of programming specialists, introduced and championed the structured programming methodology.

However, the origins of structured programming and the reasons for its quick adoption can be traced to earlier discontent in the computing field.

The years immediately following the advent of automatic coding systems should have been a fruitful time for programming specialists. Automatic coding systems were, as shown in the last chapter, a tool to increase the sophistication and scale of programmable tasks. They accomplish this by providing English-like notation, subsuming routine clerical housekeeping tasks like symbolic addresses and containing subroutine libraries. [180] This should have eased the tedium of programming. Automatic coding systems transfer the burden of tracking the intricate and repetitive details from the programmer to the software. This liberates the vocational programmers and programming specialists to work on the logical design and implementation of the program.

Yet throughout the 1960s, a close reading of the literature reveals an unexpected, systemic malaise in the discipline: concerns that the field was on a plateau and complaints about time and cost overruns on software projects are a constant theme in both the programming specialist community and the trade literature devoted to

[180] Adams, "Simple Automatic Coding Systems," 5-9.

vocational programmers. Nathan Ensmenger illustrates that, as well as the technical issues in the programming specialist literature, there was also significant concerns about the field's professionalism (job status and longevity in particular) in the vocational programming literature.[181]

Stuart Shapiro further illustrates this dissatisfaction, arguing that the root of the dissatisfaction were the problems of complexity. Shapiro argues that the dominant reason for the complexity of software is the plasticity that characterizes the technology. For example, he argues that to scale up in software was a significant hurdle, because it differed so significantly from physical sciences and technologies. Shapiro goes on to argue that structured programming was adopted with such vigor by the community because it was perceived as a solution to problems of complexity.[182] I reinforce that argument in this chapter and take it further, arguing that not only was structured programming accepted by the community because of the perception that it would resolve problems of complexity, but it was created in response to the problems of complexity and verification.

---

[181] Ensmenger, *The Computer Boys Take Over*, 167

[182] Shapiro, *Computer Software as Technology*, 111.

Priestley's work also touches on the structured programming debate. However, we differ substantially on our interpretation of this event. Priestley argues that this was a significant technological shift in programming theory, arguing "There can be no doubt that structured programming made a significant and lasting contribution to programming language design and programming practice."[183] In this chapter, I argue that structured programming was (in the way it was adopted by the programming community) a rhetorical shift, not a technological change.

Dijkstra and Peter Naur (of the earlier mentioned Backus-Naur Form) both lamented that the computers of 1962, despite being faster and having significantly more storage, were disappointing, and they were mournful of the inelegance of the design of the modern computer. [184] Naur felt that this inelegance of design was because the entire computing community was paralyzed by the drive towards efficiency.[185]

---

[183] Peter Mark Priestley, "Logic and the Development of Programming Languages, 1930 – 1975", 195

[184] "Transcript of Some Meditations on Advanced Programming Presented at the IFIP Congress Munich," 1962, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

[185] Peter Naur, private correspondence, 1961, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

There are three aspects to efficiency in computing: efficiency of hardware, efficiency of software as an element of program optimization, and efficiency of programming as a task. Naur and Dijkstra were both lamenting that the focus of hardware and software design was increased efficiency, without a commensurate interest in the elegance of the design. Naur and Dijkstra were also convinced that efficiency would be a by-product of elegant design. An elegant design, in their eyes, would be one that resolved a problem without added, unnecessary complexity. However, with the benefit of hindsight, I argue that while efficiency is not isolated from complexity, it also doesn't have a causal relationship. An elegant design (one that is not unnecessarily complex) is not necessarily the most efficient. At times, a more efficient system may be less complex; however in other instances the reverse is true.

A good example of this is virtual memory. A computer has a limited amount of primary memory (the equivalent of RAM in a PC), but it has a lot of secondary storage (the equivalent of the hard drive). A machine with virtual memory moves the least used and least critical memory elements from the primary memory to an area on secondary storage medium – a virtual version of the primary memory. Virtual memory significantly adds to the complexity of the computer system, but correspondingly increases the efficiency of the system.

Dijkstra and Naur's critiques raise questions about the role of efficiency in computing during this time period. As hardware speeds and the amount of available memory increased (or, put another way, as computers became more efficient) the types of industries and the different purposes people wanted to apply computers to correspondingly increased. Luminaries in the field, like Joseph Carl Robnett (J.C.R.) Licklider, were advocating for interactive computing with time-sharing capabilities and telecommunications. These developments, both in hardware and in the perception of what the computer should be applied to, increased the scope and size of the software applications required to manage such tasks. These new systems required that software run more efficiently, i.e. do more tasks in less time.

SAGE (Semi-Automatic Ground Environment) is a good example of a large scale, real time, interactive system. SAGE was an air defense system, designed to protect the United States from long-range enemy bombers. The SAGE system sent information from geographically distributed radar stations over telephone lines and processed it at a central location. The processed information from the radar stations was displayed on a cathode ray monitor. Operators were then able to easily interpret these displays, assess threats and

contact both the command and the operations centers. [186] It was necessary for the SAGE system to operate in real time, because if the radar information wasn't processed and relayed back to the operator in real time, the operator would be assessing threats that had already passed, negating the efficacy of an early warning system.

This type of real time system differs significantly from programs that process static data sets. For example, large companies that are using a computer to automatically bill their clients at the end of the month have a static data set. The computer sequentially totals each bill and prints an invoice. While the company may still prize speed, the data being processed is static. If the first bill being processed takes longer to be totaled and printed than expected, the total on the next bill isn't going to change while waiting in the queue to be processed.

In the 1960s the work on real time systems increased the complexity of programming theory. The magnitude and sophistication of the programs being created increased. For example: real time programs need to control concurrent access; they must keep a number of connected systems up-to- date with changing situations.

---

[186] John F. Jacobs, "SAGE Overview," *IEEE Annals of the History of Computing* 5, no. 4 (1983): 323-329.

Furthermore, the program needs to work under strict time requirements, which may result in rejecting the algorithm that fits the solution most naturally.

Success, in this case, precipitates demand. As these large real time systems became operational, there was demand from different fields and industries to apply the power of the computer to novel areas of application. However, this success had consequences: a large upward peak in the number of vocational programmers, the software crisis, and even the unbundling debate. As more computers are purchased, more software is required.

In this sense, the successful quest for efficiency drives the field. However, I argue that, unlike complexity and verification, in computing increased efficiency is a general concept, despite being quantifiable. Striving for a more efficient computer or software system is like telling people that they are striving for a "better" computer or software system. This never-ending quest for increased efficiency may be a stimulus for the industry, but it is not driving specific technical changes. Alternatively, complexity and verification drive specific changes to the technical practice of writing software.

In the commercial programming community, complaints similar to those of Dijkstra and Naur were being expressed in different ways.

T.B. Steel argued at the 1962 ACM National Conference that large scale, real time systems introduced such complexity to the programming process that existing automatic coding systems (like FORTRAN, ALGOL, and COBOL) were unsatisfactory as vehicles for these types of applications.[187] But why would automatic coding systems be unsuitable for large scale, real time systems? What were their limitations? To explore these questions I am using SABRE, the American Airlines reservation system, as a concrete example of a large scale, real time system.

SABRE

By the early 1950s the existing reservation system for air travel was becoming overwhelmed as the number of travelers significantly increased. American Airlines pursued a number of electromechanical solutions to streamline their reservation process and increase the accuracy of seat assignments. These solutions turned out to be ineffective. There was a high level of inaccuracy and they were also becoming increasingly expensive because of the large number of human operators needed to carry out a reservation.[188] After a chance

---

[187] T.B. Steel, Jr., "Automatic Programming and Compilers III: Languages and Real Time Information Processing," in *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers* (New York: ACM, 1962), 90.

[188] Duncan G. Copeland, Richard O. Mason, and James L. Mckenney, "SABRE: The Development of Information-Based Competence and Execution of Information-Based Competition," *IEEE Annals of the History of Computing* 17, no. 3 (1995): 30-33.

meeting between two executives (C.R. Smith from American Airlines and Blair Smith from IBM) American Airlines and IBM negotiated for 6 years to computerize the reservations system. In 1959, IBM and American Airlines began a collaborative effort to create a computerized reservations system.[189] That system was SABRE, the Semi-Automated Business Research Environment, a play on the SAGE acronym and IBM's last foray into large scale, real time systems.

American Airlines needed a system that would automatically and accurately track the number of seats on each of their flights. Over- or underselling a flight had significant consequences for American Airlines. Overselling a flight (booking more passengers than seats) could ultimately result in penalties from the Civil Aeronautics Board. Overselling seats also had customer satisfaction ramifications. Underselling flights resulted in a less profitable flight. There were also customer service ramifications if a flight was (mistakenly) sold out, because it would drive customers to a competitor. To accurately track seats American Airlines needed a system that functioned in real time. The system would need to disseminate an accurate seat count to AA sales locations. Sales locations then needed to be able to make a change to the seat count and that updated count needed to be made

[189] Duncan G. Copeland *et al.,* "SABRE," 34.

available to other sales locations in real time, to prevent overbooking the flight.[190] At this time, these type of large scale, real time programming systems, which could communicate over long distances, were virtually unknown in the commercial sector. SAGE, the air defense system, and ERMA (Electronic Recording Machine-Accounting), the banking system designed for the Bank of America, were the two most comparable projects.

To create this automatic, real time reservation system, IBM proposed to use a combination of teletypes for communication and IBM computers to process the information. In the SABRE system, all American Airlines sales locations would have a teletype. These teletypes would send inquiries and booking requests and receive responses automatically, while these requests would be fulfilled by the central computer, which tracked the number of available seats on the aircraft. An image from the Copeland, Mason and Mckenney article, "SABRE: The Development of Information-Based Competence and Execution of Information-Based Competition," clarifies this process:[191]

---

[190] Duncan G. Copeland *et al.,* "SABRE," 34.

[191] Duncan G. Copeland *et al.,* "SABRE," 36. Figure 1, sourced to American Airlines.

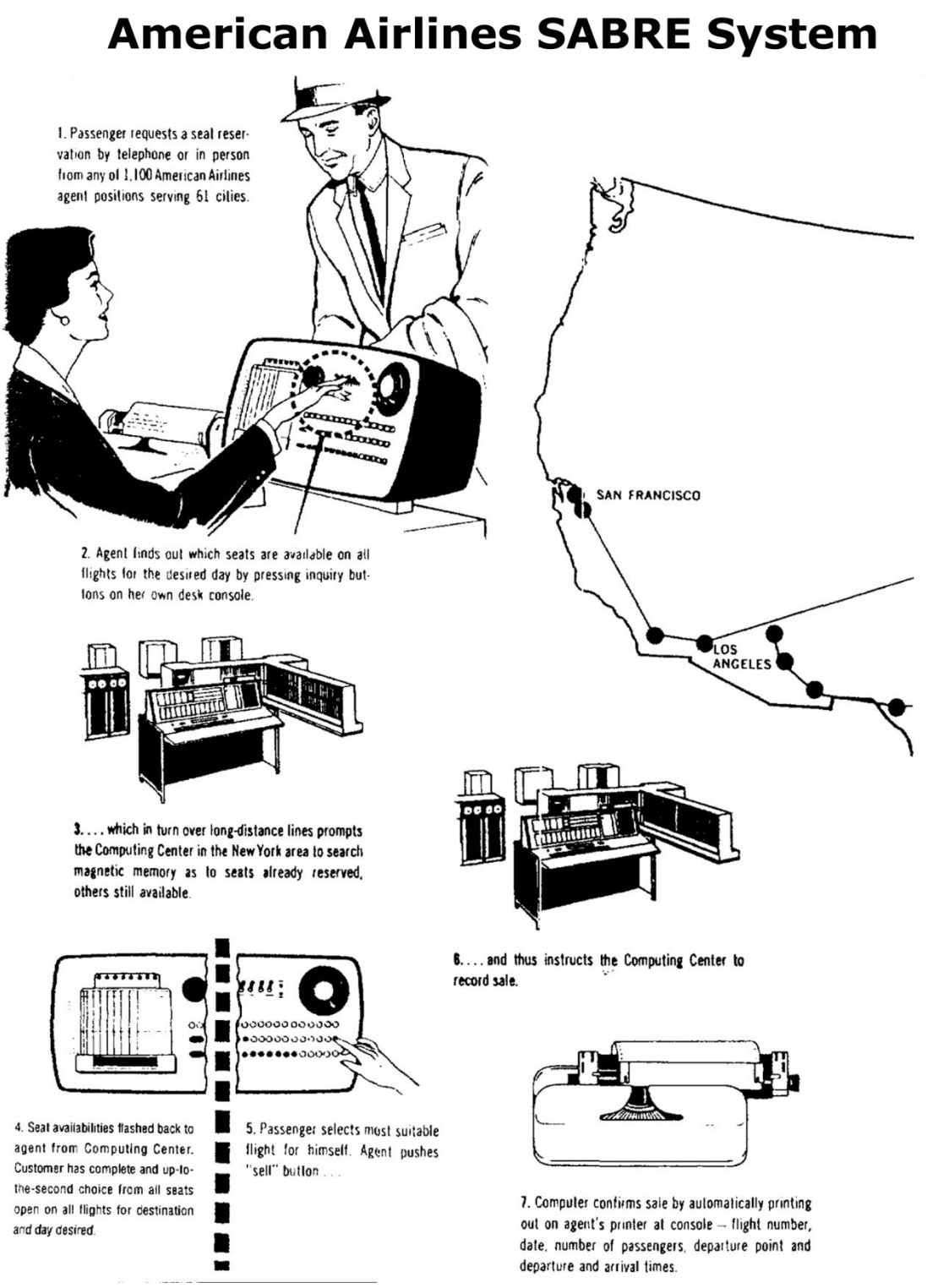


Figure 9: The Conceptual Design of the SABRE System

Coding SABRE

The coding phase of the SABRE system began in earnest in 1961.[192] Unlike at the inception of the SAGE system in the early 1950s, in 1961 high level programming languages were common. In 1960 the COBOL language definition was finalized and several compilers existed (although IBM had not yet implemented a fully compatible COBOL compiler).[193] FORTRAN was readily available on the 709, the computer being used to implement the SABRE system.[194] However, there is no discussion in the historical record about what programming language SABRE would use. Indeed, there was little discussion even about the language that was used. Based on my research of the SABRE records, there is no discussion about what programming language would be used for SABRE in any of the published papers, the IBM journals, or in the archival record that I was able to examine.

Originally, I considered that perhaps IBM simply didn't want to disclose the language, but then I discovered a brief mention of the SCAT (Share Compiler Assembler Translator) language in Robert V.

---

[192]R.V. Head, "SABRE Planning," (unpublished paper, 1962), Robert V. Head Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis.

[193] R.W. Bemer, "A View of the History of COBOL," *Honeywell Computer Journal* 5, no. 3 (1971): 133.

[194]IBM, *FORTRAN Monitor for the IBM 709*, manual, (IBM, 1959-1960), Computer History Museum, http://www.computerhistory.org/collections/accession/102678932.

Head's archive. Head was a senior planning specialist who supervised program specifications for the SABRE system. The reference to the SCAT language was in the training tools provided to American Airlines to teach their new programming cohort. I argue that the reason there is no debate over the programming language to be used in the SABRE system is because, at that time, it was simply understood that of course this complex real time system would be written in the syntax of the computer's assembly language.[195] Long prior to the beginning of the coding phase the project group had chosen, with (we must assume) minimal discussion, to move forward with SABRE using the SCAT language.

This raises questions. What was SCAT? Who developed this language? In 1956, Share (IBM's user group) began defining an operating system for the IBM709. The implementation of the Share operating system was the responsibility of the Applied Programming Department of IBM. The Share operating system (SOS) was a complete, integrated system designed to facilitate communication with the 709. SOS provided a symbolic assembly language, which was common across the four responsibilities of the operating system:

---

[195] M. Rothstein, "American Airlines 709 – 7090 Training Problem," (unpublished document, April 6, 1960), Robert V. Head Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis.

1. The Share Compiler Assembler Translator (SCAT)
2. The automatic input/output system
3. The debugging system
4. The supervisory control program: the master program that maintained the computer's continuous operation

This language was heavily based on the symbolic assembly language of the 704 (SAP). The SOS language was a conventional symbolic assembly language that used punched cards. Each instruction has 4 elements: operation, address, index register, and a counting constant. To illustrate, below is a program that would sum the numbers in locations 100 – 109.

Location	Operation	Variable	Comment
Start	CLM		Clear Accumulator
	LXA	L10, 1	Set index register
	ADD	110,1	Accumulate sum
	TIX	Current, - 1, 1, 1	Test and step index
	STO	ANSWER	Store answer
	HTR	Current +3	Transfer Out
L10	HTR	10	Constant 10
ANSWER	HRT		ANSWER store

SAP Program to find the sum of numbers in locations 100 – 109
(Source: Share Operating System for the IBM 709[196])

[196]IBM, unpublished memo, April 6, 1960), Robert V. Head Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis

SCAT extended the SAP language and was used to solve a long-running problem with symbolic assembly languages.

One problem with using symbolic assembly languages (like SAP) was that the amount of computer time necessary to assemble the program was expensive. The machine would read the symbolic program, translate the commands into machine language and produce a binary card deck. This binary deck was the operational code used during run time. Because the assembly time was significant, debugging changes were usually made in machine language (literally, binary code) to the binary deck. When referring back to the symbolic program cards, these changes were not reflected, resulting in considerable difficulty for new, usually vocational, programmers maintaining or extending a system.

In SCAT, the original symbolic program deck is loaded onto the machine, just like with SAP. The computer then processes that program, but not just into machine language (binary code). Instead the computer produces a deck of cards that retain an encoded symbolic form as the compiled version of the program (not a binary deck). The completely assembled, binary version of the program was, at least in the SABRE system, stored either in the drum or on magnetic tape storage, depending on the priority level of the program.

Programmers can then debug and make changes directly to the partially assembled version of the program, but that deck replaces the original, symbolic deck, resulting in a single functional version of the system that both the computer and the programmer could understand. In this way, all communication with the machine was conducted in the same language.

Looking at the small assembly program provided, the SCAT programming system is less user-friendly than automatic coding systems like FORTRAN or COBOL. In the literature, we read that using this most basic symbolic programming system was necessary with large scale, real time systems. But, why was this true?

There was little written about why the large real time systems of the 1950s and 1960s did not use high level languages, but there are explanations as to why a programmer would not write systems software in high level languages. Many of the same reasons that high level programming languages are not appropriate for systems software hold true for large scale, real time systems.[197] System software are programs that manipulate the computer's hardware and provide a platform for applications. Operating systems are, for example,

---

[197] John G. Fletcher, "No! High Level Languages Should Not Be Used To Write Systems Software," in *Proceedings of the 1975 ACM Annual Conference* (New York: ACM, 1975), 209-211.

systems software. Hardware drivers are another example. System software has a number of common traits: it is frequently used by the processor, consumes much of the computer's resources, and performs high priority tasks that cannot be interrupted. These tasks include input/output routines, storage allocation, file access, and scheduling. Large scale, real time applications perform many of these same tasks, with similar requirements, during their execution.

Symbolic assembly languages are more efficient in terms of both execution time and storage requirements than high level programming languages. This is in part because of the compilation process. When a program written in a high level programming language is executed, it is first compiled into assembly language. The compiler chooses the assembly language routines that best fit the instructions of the high level program. The computer then runs the assembly language version of the language, transforming it into binary code.

Unfortunately, compiler programs will not always choose the most efficient algorithm of translation. Alternatively, when writing in assembly language, the programmer must figure out the most efficient algorithm, and that algorithm is translated directly into machine code (binary code). Because real time systems use so many resources, both in terms of execution time and storage, the compiler's choices

may not produce sufficiently compact or high speed assembly language code.

The primary reason to use a high level programming language is because it provides the most convenient method of describing and manipulating the data and instructions of the program. For instance, if the programmer was planning to manipulate numbers, FORTRAN was the best language for that purpose. In real time systems programming, programmers were often manipulating the computer itself – determining orders of access for files, polling remote terminals, and creating or editing records, so the language that most conveniently describes and manipulates the computer itself is assembly language.

Moreover, many high level languages actually restrict the programmer from accessing the most basic elements of the computer to protect the underlying system. In real time systems programming, often programmers do need to change the underlying function of the computer system. For example, when using floating point arithmetic, there are often two methods available (unnormalized and normalized) and in each method the decimal can be either rounded or chopped. If the program requires a significant amount of flexibility when addressing numerical data, a high level programming language (during

this time) would usually only provide one implementation of floating point arithmetic, despite the underlying computer hardware having more flexibility.

To illustrate a limitation of a high level language, using a real world example, we will return to the SABRE system. In the archival record, much of the discussion in the SABRE programming group centered on how to pack data into the bytes. For example, one element of the SABRE system was to notify operators of flight delay and to adjust passenger connections accordingly. The programmers wanted to send a byte that would contain the following information: the flight code, the airport code, and a coded reason for the delay.[198] In real time systems this type of data packing is common, because the programmers needed to get the most byte for their buck when storing information in memory. The core memory space in the SABRE system was so valuable that it was necessary to store as much data as possible in each byte.

In the high level programming languages of the time it was impossible to pack information into the byte if it was in violation of the prescribed data unit. For example, the constraints of the language

---

[198] IBM, "Inventory File," (unpublished paper, 1959), Robert V. Head Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis.

may be such that it requires a single word byte, not one that would need to be unpacked. It would also be impossible to concatenate data units or to use a fraction of the unit, further limiting how the bytes were used. At this time all real time systems needed to be able to use the entirety of the machines' resources, and high level programming languages could not fulfill these requirements.

These criticisms of high level programming languages were never resolved by a "better" language. Instead, hardware improvements began to make them obsolete. As memory becomes larger (and cheaper) there is less of a need for such precise control over the memory. More efficient virtual memory becomes commonplace, further negating the need for such control. With more space and speed at their disposal, programming languages can implement more options, which solves problems like the alternative types of floating point arithmetic implementations. Furthermore, new high level languages have been designed with these issues in mind, allowing this type of low level access to the machine. The C language is a good example of this. It provides both the high level ease of syntax that programmers want with the low level access to the machine that some applications require.

Attitudes have also changed. The field of programming theory as a whole has become less focused on a single solution (like the universal programming language concept so popular during the creation of ALGOL and COBOL). It is now widely accepted that different programming applications require different languages, be that the use of assembly language for systems work or FORTRAN for scientific computation.

In the 1960s this was not true. There was a definite trend towards universal solutions. The desire for a universal programming language was expressed in the steering committees for both ALGOL and COBOL. As a result, while the criticisms of the automatic programming systems seem unrealistic in hindsight, at the time they were deeply felt failings.

COBOL was particularly singled out for this type of criticism, probably because it was additionally burdensome given it was part of the requirements for Department of Defense projects. COBOL was described at the 1964 AFIP conference as clumsy. It was also criticized because the compilers did not fully conform to the language definition.[199] Many computer scientists believed that COBOL's

---

[199] Saul Rosen, "Programming Systems and Languages: A Historical Survey," in *Proceedings of AFIPS '64 Spring Joint Computer Conference, April 21-23, 1964*, (New York: ACM, 1964), 8-10. Saul Rosen was the Chief Software Designer at Philco. He later became the Director of Purdue's Computing Center and was a recipient of an ACM Distinguished Service Award.

popularity was due to the DoD requirements that promoted COBOL, not its merits as a data processing language.[200] As early as 1961, just a year after its formal specification, COBOL was described as a blind alley by John McCarthy, the author of LISP and a significant contributor to the artificial intelligence community. McCarthy was, like Dijkstra and Naur, intent on creating a mathematical theory of computation. The COBOL committee purposely created an English-like syntax, but McCarthy argued that English is not well suited to formal (mathematical) descriptions of procedures.[201]

State of the Field

When reading the literature, this general air of disgruntlement of programming specialists seems at odds with the enormous progress that the historian sees in these early days of the technology. In the 15 years between the specifications of the EDVAC (the first description of an electronic, stored program computer) in 1944 and the complaints seen in the early 1960s, the computer had gone from being the subject of experiment to an integral part of numerous industries and government departments.

---

[200] Ibid., 9.

[201] John McCarthy," A Basis for a Mathematical Theory of Computation, Preliminary Report," in *Western Joint IRE-AIEE-ACM Computer Conference, May 9-11, 1961,* (New York: ACM, 1961).

Between 1944 and 1960, the processing power and memory capacity of computers had increased dramatically with the integration of new technologies like magnetic core memory and transistors. Peripheral technologies, like IBM's 1403 chain printer created for their 1401 computer, SAGE's light pen, or IBM's RAMAC (Random Access Method of Accounting and Control — a magnetic disk used for secondary storage) were being developed and integrated into new systems. New ways of thinking about computing, like time sharing and the use of telecommunications, were being developed and implemented.

As the hardware improved, so, too, did the software. Software applications were being created for a wide variety of purposes. Computers were being used for everything from simple batch processing tasks like managing payroll and sales records at Westinghouse, calculations for the Census Bureau, or automated bank record keeping with the introduction of the ERMA system.[202] The introduction of real time software systems like SAGE and SABRE challenged the limits of the existing technology and pushed the boundaries of the programming discipline. These real time systems were integral to the interactive computing concept that eventually led

[202] ERMA's significant contribution is the automatic handling of checks, with the development of magnetic ink and optical character reading (OCR) technology.

to the personal computer. Automatic programming techniques were flourishing, both in the area of generalized languages like FORTRAN, COBOL, ALGOL, and in the boutique languages like LISP. Discussion and debate about programming theory was a frequent topic in the literature of the ACM.

"Computer science" was being accepted as an academic discipline, first appearing in the undergraduate syllabus of many colleges, as part of other departments including mathematics and electrical engineering. Purdue was the first university to offer a degree in computer science, beginning in 1962. 1962 was also the year that the ACM created a committee on computer science education. By 1963, it was estimated that there were a dozen universities offering a computer science degree, a number that kept growing over the next decades.[203]

In light of these signifiers of success, I argue that within the field there is agreement about the future direction of computing in the 1960s in a way that did not (and could not) exist in the 1950s. Broad examples of this can be seen when we contrast the state of the field in the 1950s with the 1960s. In the early 1950s the computer manufacturing industry was unstable, with a large numbers of

[203] G.K. Gupta, "Computer Science Curriculum Developments in the 1960s," *IEEE Annals of the History of Computing* 29, no. 2 (2007): 45-52.

companies entering, and then quickly departing, the field. By the 1960s the computer manufacturing industry had stabilized, with IBM and seven smaller companies remaining. In the 1950s debate was about the role of automatic programming systems, while in the 1960s the debate had moved on to *which* automatic programming systems should be used. Real time processing systems in the 1950s were thought to be applicable only in the most crucial of cases or largest of industries. By the 1960s, real time systems were becoming more common and were being used by smaller industries and organizations.

Increasing Complexity in the Field

Despite these indications of growth and agreement about the future of the field, these elements of success are also the underlying cause of the general dissatisfaction that we've seen in the literature. I argue that this dissatisfaction was the catalyst for the "Structured Programming Revolution." The tension between prior success and the difficulties faced by those now working on the boundary of the field created a sense of urgency to overcome problems and reach another breakthrough to further the field.

The success of computerizing so many areas of application increased demand for computers and the software that drove them. This demand for new programs contributed to the acceptance of

automatic coding systems as a method of accelerating programming projects, but the success of automatic coding systems led to demand for even more new vocational programmers. As a result, the ACM and IBM both engaged in concerted efforts to increase the number of both vocational programmers and programming specialists. Their methods focused on training techniques and working with universities to increase the number of "computer science" departments. During the late 1950s and the early 1960s IBM donated 50 computers to universities and the ACM made numerous syllabi recommendations.[204]

The focus on attracting new vocational programmers opened up the field to a greater number of people, people with diverse backgrounds and differing understandings of the computer. It is estimated that the number of people involved in the computing industry increased nearly three-fold from an estimated 10,000 computing professionals in 1955 to 30,000 in 1960.[205] However, despite this flood of programmers into the field, the software crisis

---

[204] Gupta, "Computer Science Curriculum Developments," 3, 5.

[205] T.A. Dolotta, et.al., *Data Processing in 1980 – 1985: A Study of Potential Limitations to Progress*, (New York: John Wiley and Sons, 1976). This statistic is higher than that used by Martin Campbell-Kelly in his seminal work, *From Airline Reservations to Sonic the Hedgehog*, who states on page 39 that in 1959 there were 1400 programmers in the United States. Campbell-Kelly later uses a statistic from *Business Week* arguing that there were 120,000 programmers in 1966. Ensmenger argues that these seven hundred programmers were 3/5ths of the total industry – 1,166 programmers. (p 60) The statistic I am using included more than just programmers, but it is based on the census figures. Regardless of the actual numbers, the point of the statistic is to illustrate the very limited number of people involved in the industry in the late 50s/early 60s versus the veritable explosion of the industry by the late 60s.

continued. This crisis was recognized officially at the 1968 NATO Software Engineering Conference. Ensmenger argues that at this time the rhetoric of crisis became one of the defining features of the software industry.[206]

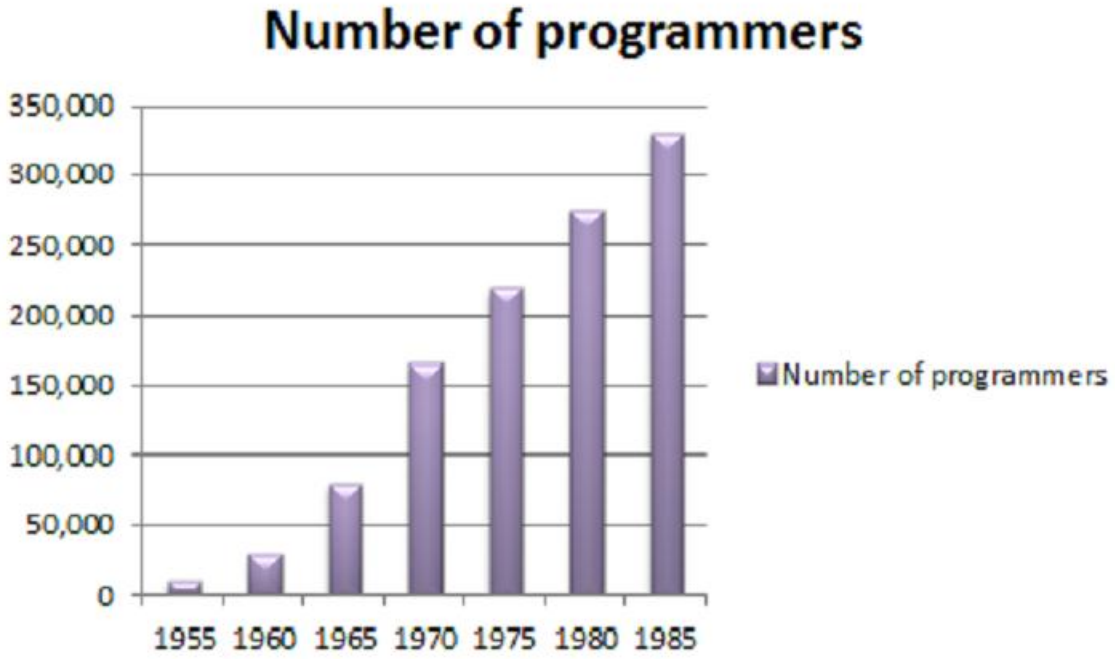


Figure 10: Increasing number of programmers over time
Source: T.A. Dolotta, et.al., "Data Processing in 1980 - 1985 A Study of Potential Limitations to Progress."

This increase in the sheer number of vocational programmers, and the change they brought to the face of the profession, increased the complexity faced by programmers, both vocational programmers and programming specialists. Throughout the 1950s the programming field, including both the vocational programmers and the programming

[206] Ensmenger, *The Computer Boys Take Over*, 195.

specialists, was small. The background of most of these programmers was similar: they entered the field through mathematics and physics degrees. Many of them were contracted to work on the same projects. SAGE is one example of this. At one time, SDC (the company in charge of the programming effort for the SAGE system) was jokingly referred to as the programmer's university because they trained 700 programmers for the SAGE effort.[207] I would argue that these programmers were not vocational programmers. Rather, they were all programming specialists. The small size and shared background of these programming specialists contributed to a base of tacit knowledge. The later flood of new, vocational entrants into the programming field did not have this tacit knowledge. Without this underlying body of knowledge communication between programmers became less organic and the work needed to be more structured.

Furthermore, these vocational programmers were faced with increasingly sophisticated large scale software projects. The more complicated real time systems were becoming more prevalent. New and different industries were clamoring for computerization. Alternative conceptions of computing, like time sharing, interactivity

---

[207] Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine,* 2nd ed. (Boulder: Westview Press, 2004).

and telecommunications were emerging as requirements for new computer systems.

The most famous of these sophisticated, large scale software projects wasthe IBM System/360. Announced in 1964, the System/360 was a family of plug-and-program compatible computers, which covered a range of applications needs, in both the scientific and business markets and for both small and large scale concerns. The complete range included six processor models and forty-four different peripheral devices. This family of compatible computers was based on the new SLT (solid logic technology) module, electronic circuits that replaced the earlier transistor models.

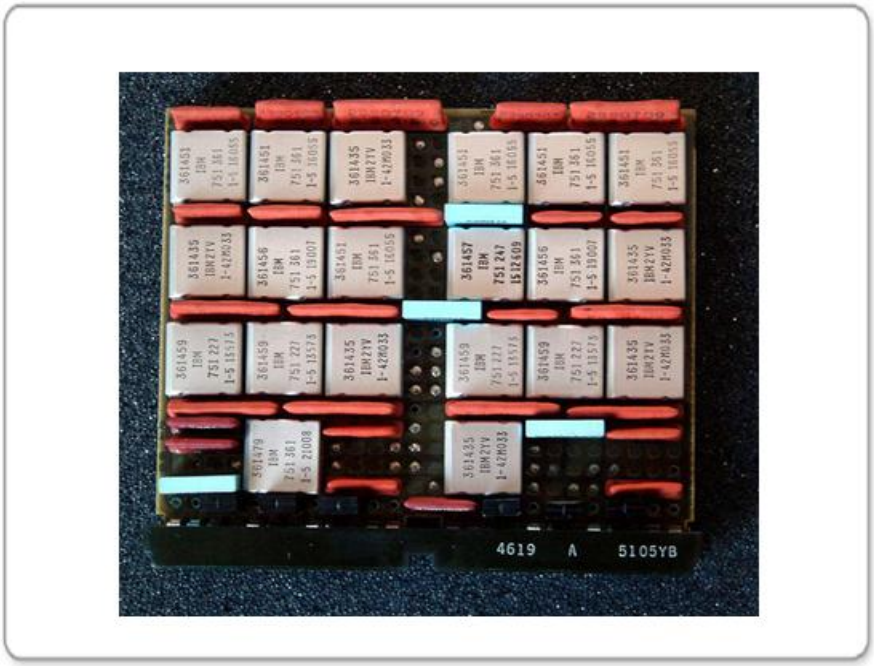


Figure 11: SLT Card from the IBM 1130 circa 1965
Source: Wiki Commons, Photographer: James Berlin

Not only was the System/360 family based on new hardware technologies, producing code compatible computers over such a large range was also a new technological feat. The System/360 architecture would use the control store concept first proposed by Maurice Wilkes of subroutine fame. A control store is a read only storage device that can be programmed to manipulate the underlying computer architecture. In this way, a larger instruction set can be used, without hardwiring it into the logical design of the computer. These instructions are referred to as microinstructions and are conducted in microcode. Originally included as a way of expanding the scope of each machine so that it could be easily optimized for either business or scientific applications, without having to hardwire the entire instruction set to include instructions applicable to both types of applications, the efficient use of microcode was a technological breakthrough that would eventually save the System/360 project.

It had been decided at the inception of the System/360 project that software written for one machine in the family must be able to run on any other System/360 machine.[208] The logic behind this was that the system would benefit from the same sort of economies of scale found in mass production. Originally, it was assumed that this

[208] The IBM System/360 was not, however, backwards compatible with existing IBM computers.

compatibility could be completed using emulation, simulation or translation technology (where software of one computer is used to simulate the logical design of another computer) that was readily carried out using assembly language. It was soon recognized that the performance of any of these methods would not be efficient enough – particularly for the important system software, the operating system that would run peripheral devices, allocate memory and storage, and manage other tasks crucial to the operation of the computer. This major design flaw left the System/360 open to serious critique from internal opponents of the project.

The project group was saved by a last minute technical accomplishment in the use of microcode to improve the simulation process. Microcode is a layer of hardware-level instructions that can be used to mimic the architecture of the machine, allowing for binary level compatibility between computers with different architecture. Microcode functions through the read only control store that had already been included in the System/360. After this breakthrough, it was decided that IBM would produce a single operating system, OS/360, and supplement that operating system with three special operating systems (BOS – basic operating system, TOS – tape operating system, and the more complex DOS – disk operating

system) that were optimized for the differing ranges of computers and peripherals.

The software for computers of the System/360 was, theoretically, both upwardly (programs written for the smallest computer could run on the largest) and downwardly (programs written for the largest computer could run on the smallest) compatible. Upward compatibility was well resolved with the introduction of microcode, but downward compatibility was, arguably, never really resolved. There were a number of documented exceptions in the smaller processors that would prevent them from running some software programs written for the larger processors. Moreover, even software that complied with the strict programming definitions required for downwardly compatible software would run painfully slowly on the smallest machines. However, even with these limitations, the compatibility provided was a significant development in computing. Further, the use of microcode to abstract the instruction set from the hardware, developed to provide this compatibility, was a momentous breakthrough in computing.

In addition to the sophisticated challenges presented by the compatibility requirements, during the project team's work on the System/360, changes in the basic understanding of how computing

should be conducted were sweeping through the computing community. As we have seen, while most processing done at this time was in a stacked job mode (essentially batch processing where each program is completed before another one was started) interest was growing in multiprogramming and in interactive applications where people could interact with the computer from terminals in remote locations. Even before the original System/360 operating systems were complete, the IBM programmers were already hard at work on the multiprogramming version of DOS. With this added functionality, DOS became the most widely used operating system in the world. Originally conceived as the supplemental operating system for the higher level, disk-based System/360 machines that had at least 16K of memory, DOS was re-released 26 times between 1966 and 1971, when it was finally decided that no more new versions would be created.[209]

The System/360 project has been called a "$5 billion gamble," because IBM had essentially bet the company on this product family. The System/360 focused on promoting modularity and compatibility, allowing companies to add modules of processing power and take their software with them when finally upgrading to a faster, more costly

---

[209] Emerson Pugh, *Building IBM: Shaping and Industry and its Technology* (Cambridge: MIT Press, 1995), 263-300.

system. Systems would still be tailored to the individual needs of the customer, but those systems could be assembled from pre-existing modules and would run existing programs. This was thought to provide IBM with the kinds of economies of scale they had been able to achieve in their electro-mechanical punch card systems, and that had eluded them in the new programming field.[210] System/360 was a phenomenal business success, helping IBM to recapture their market dominance and boosting their 1965 63% market share to a 71% share in 1967.[211]

However, internally the System/360 was not seen as an unmitigated success. Instead, it was seen as a cautionary tale. Manufacturing of components for the System/360 had been fraught with difficulties, stemming from the importance of quality control with the still-fragile electronic components. Software had both time and cost overruns, leading to repeated reorganization of the software division. Moreover, the software didn't originally meet the required specifications, using more memory and with slower processing times. Two new software requirements were perceived mid-project: time

---

[210] Steven W. Usselman, "Unbundling IBM: Antitrust and the Incentives to Innovation in American Computing," in *The Challenge of Remaining Innovative: Insights from Twentieth-Century American Business*, eds. Sally H. Clarke, Naomi R. Lamoreaux, Steven W. Usselman (Stanford: Stanford University Press, 2009), 263-264.

[211] Steven W. Usselman, "Public Policies, Private Platforms: Antitrust and American Computing," in *Information Technology Policy*, ed. Richard C. Coopey (Oxford: Oxford University Press, 2004), 105.

sharing and multiprogramming. While multiprogramming was eventually implemented, time sharing on the System/360 was never implemented in a useful manner.[212] The perceived economies of scale never truly eventuated at IBM in programming software and the costs overruns for software were unexpected.

The commonly perceived solution to this growing complexity was increasing the number of vocational programmers working on each software application and correspondingly increasing the management structure surrounding the System/360 project. Fred Brooks makes it clear in the *Mythical Man-Month* that this was the perceived solution and, moreover, that this solution usually failed. In 1975 Brooks demonstrated that an increased number of programmers working on a programming project increases the managerial tasks and complicates communications. Larger programming projects become more difficult for the single programmer to envision as a whole, both because of the scope of the project and because the "divide and conquer" managerial techniques that are used to run the project can disguise the ultimate goals.[213]

---

[212] Steven W. Usselman, "Unbundling IBM," 265.

[213] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Reading: Addison-Wesley, 1975).

Nor was Brooks the first participant to recognize this problem. As early as 1960, William Orchard-Hays foreshadowed Brooks' classic book, demonstrating that the community of programming specialists was acutely aware that their field was suffering because of this increase in complexity:

> The standard solution to the problem of "getting on the air" with a big project has been to proliferate ever larger programming groups – by hiring, training, renting, pirating or any other means including dumping a bunch of people in to sink or swim. This method is becoming increasingly ineffective. A large number of people doing a mediocre job only adds to management's problems.[214]

The problem of complexity was also a topic of research at the 1962 ACM Conference. Generalized operating systems were becoming more common in the early 1960s. In one article, researchers linked this increased need for operating systems with the increased complexity of software applications.[215] At the 1968 AFIPS conference another paper was presented on the difficulty of maintaining large scale computer systems because of their inherent complexity.[216] The

---

[214] W. Orchard-Hays, "A New Approach to the Programming Problem," in *Proceedings of the IRE-AIEE-ACM Western Joint Computer Conference, May 3-5 1960* (New York: ACM, 1960).

[215] E.S. Page, C.C. Gotlieb "Business and Management Data Processing II: On the Scheduling of Jobs by Computer," *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers* (New York: ACM, 1962)

[216] Donald R. Fitzwater and Earl J. Schweppe, "Consequent Procedures in Conventional Computers," in *Proceedings of AFIPS '64 Fall Joint Computer Conference, October 27-29, 1964* (New York: ACM, 1964), 1065.

solution the writers presented was to create a new generalized programming environment to address these problems of complexity. Ironically, this paper appeared on the eve of the introduction of a new generalized programming environment—structured programming.

The IBM Unbundling Decision

Structured programming also appeared just prior to a significant change in the programming field. IBM announced on December 6th, 1968 that they would begin unbundling their software from their hardware.[217] Prior to the 1969 unbundling decision, IBM regarded software as part of their customer service program. IBM customers had access to virtually unlimited free classes in computer operations and programming. IBM's systems engineers were assigned to each customer account and, while those engineers were supposed to advise the customer's programmers, in actuality they often worked on the customer's applications. IBM classified these services as a marketing expense.[218]

In 1968, IBM became the subject of a Department of Justice antitrust suit alleging that IBM was a monopoly. These allegations had a number of parallels with the successful suit of the 1950s that

---

[217] Burton Grad, "A Personal Recollection: IBM's Unbundling of Software and Services," *IEEE Annals of the History of Computing* 24 no. 1 (2002): 65.

[218] Luanne Johnson, "A View From the 1960s: How the Software Industry Began," *IEEE Annals of the History of Computing* 20, no. 1 (1998): 36.

resulted in the 1956 consent decree that resulted in opening up the field for third-party peripherals for IBM machines.[219] Further, CDC (Control Data Corporation) was suing IBM on grounds that IBM's hardware practices were monopolistic.[220] CDC thought that IBM was prematurely announcing new products to prevent customers from switching to a different hardware manufacturer. ADR (Applied Data Research) had taken out a patent on their application flow charting software, Autoflow, as a method of protecting themselves against IBM's practice of bundling software.[221]

In addition to these legal troubles, IBM's software costs were significantly increasing. Steven Usselman illustrates this in "Public Policies, Private Platforms: Antitrust and American Computing." The IBM System 360 was very successful, but in the short term, the costs of producing the System 360 software were significant. In the 1950s software costs were 4% of IBM's engineering budget. By the mid-1960s, this had increased to 60% of the budget. Moreover, other manufacturers were creating emulator systems that allowed IBM software to run on their plug-compatible machines. In 1963 Honeywell had created the aptly named "Liberator" emulator that

---

[219]Steven W. Usselman, "Public Policies, Private Platforms," 97-120.

[220] Burton Grad, "A Personal Recollection," 65.

[221] Luanne Johnson, "A View From the 1960s," 36.

mimicked the functionality of the IBM 1401.[222] In response to these pressures, IBM chose to price their software separately from their hardware.

While the software products industry was established before IBM chose to sell its software separate from the hardware, the decision had significant ramifications for the field. IBM's unbundling decision inadvertently helped to legitimize the concept of paying for software to prospective software clients. This accelerated the growth of the industry by opening up the software products market to independent vendors.[223]

By opening up the market, the unbundling decision continued to drive the need for software. Not only were there more companies supplying the need for software and therefore increasing the attractiveness of computerization, companies could now purchase "off the shelf" software. However, those "off the shelf" applications were not the same as the shrink- wrapped PC software that we imagine. These commercial "off the shelf" applications needed to be installed and maintained by people familiar with the still-new technology. Furthermore, as Thomas Haigh notes in his article, "Software in the

---

[222] Steven Usselman "Public Policies, Private Platforms,"108.

[223] Luanne Johnson, "A View From the 1960s," 36.

1960s as Concept, Service, and Product," these applications did not meet all the needs of a company. Packaged software was now being purchased from an independent vendor and then manipulated either by in- house programmers or software services companies.[224] That "off the shelf" software was not yet fully defined and companies were still striving to find a customized or customizable solution is supported by Steven Usselman's report that the first significant increase in revenue after the unbundling decision occurred not in software sales, but in software services.[225]

Increasing Concerns About Accuracy

Corresponding with the increasing complexity facing the vocational programmer and the programming specialists, there were also increasing concerns about program accuracy. Program verification had historically been the responsibility of individual programmers or a specialized group of programmers in charge of creating the tests and carrying them out. For example in the Cape Cod phase of the SAGE system, individuals were tasked with their own testing.[226] Yet, by the time of the much larger full implementation of

---

[224] Thomas Haigh, "Software in the 1960s as Concept, Service, and Product," *IEEE Annals in the History of Computing* 24 no. 1 (2002): 9.

[225] Steven W. Usselman "Public Policies, Private Platforms," 110.

[226] Henry S. Tropp, Herbert D. Bennington, Robert Bright, *et al.,* "A Perspective on Sage: Discussion," *IEEE Annals of the History of Computing* 5, no. 4 (1983): 380.

the SAGE system, a specific department of vocational programmers was in charge of creating and governing the tests.[227] As the programming field matured in the 1950s, verification became a research topic, specifically within the sub-discipline of artificial intelligence.

Heavily mathematical and theoretical in nature, participants in the work on program verification were originally moving towards creating an automatic program to conduct program verification. The first automated program verifier was created by James King and supervised by Robert Floyd of Stanford University in 1970. King combined techniques creating a "special purpose theorem prover for establishing the validity of expressions over integer variables which was developed as part of a program verifier."[228]

However, by the early 1960s AI researchers' interest[229] in program verification was waning, as they moved on to pursue other promising avenues. However, in programming theory, program verification was becoming a hot research topic. Verification techniques transitioned from automatic to the theoretical, pursued by

---

[227] Herbert D. Benington, "Production of Large Computer Programs," *IEEE Annals of the History of Computing* 5, no. 4 (1983): 357-358.

[228]James C. King and Robert W. Floyd, An Interpretation-Oriented Theorem Prover Over Integers," in *STOC '70: Proceedings of the Second Annual ACM Symposium on Theory of Computing, May 4-6 1970*, (New York: ACM, 1970), 169.

[229] Program verification was subsumed into the security field, changing the nature of the work.

programming specialists as part of the agenda to produce a formal, mathematical theory of programming.

The classic papers linking program verification and the use of mathematical proofs with program design were written in the 1960s. In 1962, McCarthy linked the ability to write accurate programs with mathematical proofs. McCarthy wanted the programming community, both those in programming theory and vocational programmers, to prove prior to coding that a program meets the required specification, both mathematically and with a computer program. To do this, McCarthy felt that the system of doing this needed to be defined more formally.[230]

In 1962, McCarthy was still thinking in terms of automated verification. By 1966 Naur identified the link between inaccuracy in software and lack of formal verification techniques, but did not work on automated verification systems. Instead, he argued in the article, "Proof of Algorithms by General Snapshots," that there was a deplorable lack of mathematical thinking in the way programming was being pursued, which was resulting in unreliable and ineffective programs. Naur was horrified that the most vocational programmers

---

[230] J. McCarthy, "Towards a Mathematical Science of Computation," in *Information Processing 1962: Proceedings of IFIP Congress 62,* ed. C.M. Popplewell (Amsterdam: North Holland, 1963), 21-28.

weren't even aware of the poor verification procedures that did exist and called for the regular use of systematic proof procedures. [231] C.A.R. Hoare was another well-known proponent of mathematical techniques for verifying program accuracy and explained in more detail that he thought techniques first applied in geometry, like the elucidation of sets of axioms and rules of inference, could be used to prove the properties of computer programs.[232] All of these actors in program verification were to become a part of the structured programming story.

Dijkstra and the Origins of Structured Programming

After the 1968 NATO conference, when the software crisis was identified, verification became a trendy topic in programming theory.[233] With the entire community (including programming specialists, vocational programmers and management) now focused on the software crisis, and with an increasing level of concern about complexity and verification, the stage was set for the go to controversy and the rise of structured programming.

---

[231] Peter Naur, "Proof of Algorithms by General Snapshots," *BIT* 6 (1966): 310.

[232] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM* 12 no. 10 (1969): 576.

[233] Donald MacKenzie, "The Automation of Proof: A Historical and Sociological Exploration," *IEEE Annals of the History of Computing* 17 no. 3 (1995): 7-29.

It is clear that Dijkstra introduced his structured programming methodology with these two problems in mind. In a letter to Professor Howard Aiken dated in the 1950s (prior to any of his published work on structured programming) Dijkstra mentioned that he was distancing himself from work being conducted on operating systems. In the letter he asserted that operating systems were too large and unpredictable and that he was dissociating from the field because he didn't approve of the direction operating systems were moving. Instead, Dijkstra argued that the central question of the software field should be to find a method of designing sophisticated systems in a controlled manner to avoid "unmanageable complexity." He thought that this would then feed into the operating system sub-discipline, because without this method, making system software (like operating systems) for the newer, larger computers would be impossible. Writing applications without a method that ensures the simplicity of design would result in inaccurate, inefficient and unreliable applications.[234] Dijkstra was thinking specifically about the problem of complexity during his research into structured programming.

[234] Edsger W. Dijkstra to Howard Aiken, 1954 ca, Edsger W. Dijkstra Papers, 1948-2002, Archives for American Mathematics, Center for American History, The University of Texas at Austin.

Nor was complexity the only problem motivating Dijkstra. Verification was one of Dijkstra's primary research concerns throughout his career. He was particularly concerned about verification during his work on structured programming. His article, "A Constructive Approach to the Problem of Program Correctness," written in 1963, highlights this concern, arguing, "The more ambitious we become in our machine applications, the more vital becomes the problem of program correctness. The growing attention being paid to this problem is therefore a quite natural and sound development."[235]

Within his "Notes on Structured Programming" manuscript, Dijkstra appealed to these concerns about complexity and verification, arguing:

> Size, complexity and sophistication of programs one should like to make have exploded and over the past years it has become patently clear that on the whole our programming ability has not kept pace with these exploding demands made on it.[236]

Dijkstra's work on structured programming was influenced by his concerns over the problems of verification and complexity. These concerns were evident in programming theory prior to his work and it

---

[235] Edsger W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," (1963), EWD209-0, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

[236] Edsger W. Dijkstra, "Notes on Structured Programming," 1969, EWD249, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin, 10.

was these concerns that made his structured programming methodology so appealing to the community, fostering the "Structured Programming Revolution." However, the origins of the methodology began long before the publication of "Structured Programming." While many of Dijkstra's early papers dealt with the questions and solutions in Structured Programming, none had the impact on the community of programming specialists that Dijkstra had with his 1968 paper, published as letter to the editor of *Communications of the ACM*, titled "Go To Considered Harmful."[237]

What is a Go To Statement and Why was it Controversial?

Dijkstra's "Go To Considered Harmful" stirred up an intense controversy in programming theory, one that as late as 1973 could still create considerable debate.[238] Dijkstra was already well known in the field when he published the letter. He had distinguished himself early in his career when he published an algorithmic solution to the problem of the shortest path. The shortest path problem is the problem of finding the shortest path between two vertices. An example of this is

---

[237] Dijkstra references his ideas to a 1966 paper (Corrado Bohm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules," *Communications of the ACM* 9 no. 5 (1966): 366-37) that demonstrated the importance of flow charts to the accuracy of programs, and insinuates the problem of the go to statement, because it can't be easily included in a flow chart, but this interpretation of the paper was not controversial and got little attention from the wider community.

[238] Lawrence R. Clark, "We Don't Know Where to GOTO if We Don't Know Where We've COME FROM. This Linguistic Innovation Lives Up to All Expectations," *Datamation*, (1973) http://www.fortran.com/come_from.html

finding the shortest distance between two geographical positions. Dijkstra was charismatic in person, but an acerbic writer, who was convinced of his own correctness. Despite being abrasive at times, Dijkstra had garnered a significant following and the support of a number of influential colleagues through his extensive personal correspondence.

The controversy surrounding Dijkstra's original letter was motivated, at least in part, by the phrasing and tone of his letter. The letter brooked no disagreement, assuming that anyone with knowledge of the field would agree. Moreover, the style of writing Dijkstra employed intimated that inexperienced or sloppy programmers were the only programmers dependent on a construct widely used at the time in both commercial and academic programming. Instantly, anyone who disagreed with the premise that programming should be conducted without the go to statement was labeled as inexperienced or sloppy. Nor was Dijkstra's premise without measurable drawbacks, as pointed out by Martin Hopkins at the 1972 ACM annual conference.[239] Despite Dijkstra's claims, go to statements could be used responsibly to produce functional programs.

---

[239] Martin E. Hopkins, "A Case for the GOTO," in *ACM '72: Proceedings of the ACM Annual Conference - Volume 2*, (New York: ACM, 1972), 787-790.

To fully understand the ramifications of the controversy, we first must ask ourselves, what is a go to statement and what was it used for? A go to statement is a language construct that allows the programmer to force the program to "jump out" of the main body of the program and proceed to a line later (or earlier) in the code. The value of this statement is in its ability to exit out of the body of the code and perform a different function. On first blush, this is a very useful language construct. However, the fallout of this is that the go to statement may be unpredictable.

A more practical (if simplified) example can be drawn from the working of a website. If you wanted a website to display a different image, depending on who was logged in, while keeping the overall theme the same, you could:

1. Load the bulk of the webpage
2. "Go To" the login information and retrieve the image based on who was logging in
3. Return to the bulk of the page and display the individualized image

However, if the login information was not returned in a way that was usable, for example, if it returns an unexpected value (perhaps no image is associated with the login) the program may never return to step number 3 and load the rest of the page. To the user, it would

appear that website had malfunctioned and not displayed the completed page. While this is a trivial example it does illustrate how there may be unforeseen results when using a go to statement without careful consideration of all possible outcomes, or, more problematically, when those outcomes become too numerous to predict. For example, even in the SABRE system, a system designed and implemented 5 years prior to Dijsktra's paper, it was estimated that there were over 1000 paths a customer request to book an airline seat could travel through before being finalized. By 1968, large software systems had become more sophisticated, resulting in even more paths to completion.

While the rhetoric about go to statements has been subsumed into most programming curricula and is one of the origin stories of computer science (making the outcome of the debate seem almost preordained), at the time it was a new problem and the conclusion was far from forgone. Go to statements are useful. When properly governed by the right clauses, the go to statement is no more difficult to predict than any other jump.

However, while Dijkstra was concerned about sloppy programming, the more significant reason for his criticism of the Go to statement can be traced to his concerns about verification. Dijkstra

had been pursuing an agenda of formal mathematical theories of computation throughout his career. Part of this agenda was the ability to mathematically verify a program's accuracy. Dijkstra wanted to see a more widespread adoption of what we would now identify as a call stack (which he called a mathematical indice).[240] He believed that this would improve the community's ability to verify programs.

A stack in computer science is simply a last in, first out data storage structure.

[240] The stack concept in computer science has a rich history of its own. The concept had been around since the 1950s, became more popular during the discussions surrounding ALGOL, and became widespread after the 1960s. For more information about the history of the stack concept, a good resource is: Sten Henriksson, *A Brief History of the Stack*, presented at The SIGCIS "Michael Mahoney and the Histories of Computing(s)" workshop, held at the Hilton Hotel in Pittsburgh, Pennsylvania, on October 18, 2009.

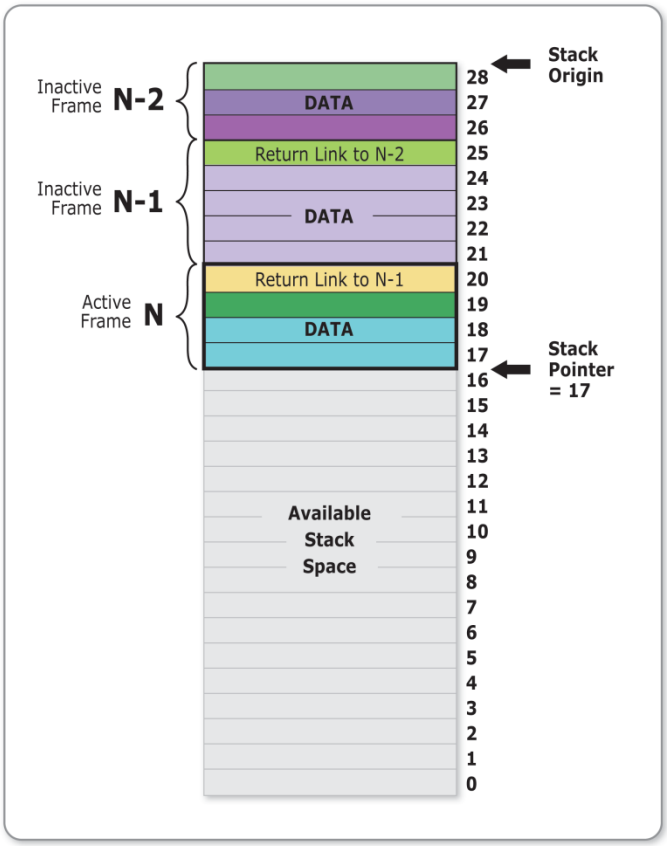


Figure 12 Graphical Representation of a Stack

A call stack is an array of statement pointers (like an index of every statement that has been called by the program). The number of statement pointers corresponds directly to the number of active procedures at any given point in the execution of the program.

Because a go to statement exits the main body and re-enters at indeterminate locations, it would significantly (if not impossibly) complicate the implementation of a call stack, making it impossible to quantifiably index the procedure calls conducted by the program.

Dijkstra approached the problem of the go to statement as a barrier to the adoption of mathematical verification. Other actors that argued against using the go to statement tended to focus on the more pragmatic day to day problem of reducing errors in software applications. However, at their essence, both Dijkstra and his more pragmatic supporters were arguing that the unbridled use of go to statements allow programs to "get lost" during their execution in ways that the programmer cannot predict.

Possibly disingenuously, in his letter to the editor about the problems with the go to statement, Dijkstra mentions little, if anything, about an alternative to using the go to statement. He suggests that go to statements should be used only in conjunction with clauses, for example:

```
GoTo with Clauses
{
*/initiate variables etc. */
        If X;
                then GoTo Y;
        else;
                GoTo (Exit);
}
```

but that is really the extent of his explanation. Moreover, this concept, that go to statements should only be used carefully and as part of a

clause, was not a new concept, as later discussion of Knuth's responses to Dijkstra will make clear.

I argue that Dijkstra was disingenuous when he did not include an alternative, because private communication illustrates that Dijkstra was already thinking about an alternative. That alternative was the structured programming methodology. From both his unpublished work (specifically EWD 241 and 245) and from his letter of clarification, "Letters to the editor: The go to statement reconsidered," one can extrapolate that, from the outset, Dijkstra had intended to replace the go to statement with the use of what we now call stepwise program composition.[241] Stepwise program composition was a significant portion of Dijkstra's later structured programming language. This alternative would change the flow of the code completely, through the use of subroutines. While I will address the stepwise program composition at more length in the following section, Dijkstra's solution to the problem Program GoTo (above) would have looked more like this:

[241] Edsger W. Dijkstra, "Letters to the Editor: The Go To Statement Reconsidered," *Communications of the ACM* 11 no. 8 (1968).

```
Program Stepwise
{
        1       /*earlier code that sets variables, conducts operations etc.*/
        2       If V > W
        3               then call (Subroutine X)
        4       else
        5               [do other important work]
}
Subroutine X
{
        22      V= V+X
        23      exit;
}
```

This is the root of the change from using the go to statement to using Dijkstra's stepwise program composition.

Underlying Dijkstra's work is a profound concern about the ability to verify the accuracy of software. In "Letters to the Editor: Go To Statement Considered Harmful," he states, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce,"[242] demonstrating this concern explicitly. While there were concerns about accuracy in the broader programming community, unfortunately, in "Letters to the

[242] Edsger W. Dijkstra, "Letters to the Editor: Go To Statement Considered Harmful," 147; Edsger W. Dijkstra, "Towards Correct Programs," EWD241. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin (approximate date: 1968); Edsger W. Dijkstra, "On Useful Structuring," EWD245. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin (approximate date: 1968).

Editor: Go To Statement Considered Harmful," Dijkstra did not illustrate the specific ramifications of the call stack and removing go to statements as they related to verifying program accuracy. Dijkstra did not provide a clear explanation of the relationship until he distributed "Notes on Structured Programming" in 1969. This miscommunication sparked a controversy lasting for most of a decade.

Gary Durbin, a software pioneer who eventually focused on databases, reflected on the controversy over the go to statement. Like Dijkstra, Durbin thought that the reason to avoid the go to statement was clear cut. He argued that the problem with the go to statement was that programmers could not isolate code if there was a sudden branch in the middle of the program. The programmer wouldn't be able to tell the state of the program –literally, a snapshot of the various conditions of the program, for example: which variables were still necessary, which ones were now obsolete. Durbin argued that structuring code to limit the scope of each piece of code would resolve these problems, because then the programmer could find the state of each code segment.[243] However, Durbin also supports the assertion that the controversy was sparked not by the content of structured programming but the manner in which it was addressed.

---

[243] Gary Durbin, Oral history interview by Philip L. Frana, May 3, 2002, OH 368. Washington, DC. Charles Babbage Institute, University of Minnesota, Minneapolis.

Durbin argued that, had Dijkstra promoted structured programming as a methodology, prior to the controversial condemnation of the go to statement, the benefits of the methodology would have been more readily accepted.[244]

Dijkstra's "Go To Statement Considered Harmful" letter resulted in almost immediate criticism. The following issue of *Communications of the ACM* published a letter of response from John R. Rice. His heated tone conveys the conflict that was to infuse the debate over structured programming:

> I was taken aback by Dijkstra's attack on the go to statement, which is an obviously useful and desirable statement…It is claimed that the go to statement is, in some sense, programmer dependent while the other statements are not. I am unable to understand this reasoning …I find the emotional tone of this attack as disquieting as the "scientific" analysis. How many poor innocent, novice programmers will feel guilty when their sinful use of go to is flailed in this letter.[245]

In a letter to Salton (editor of *Communications of the ACM* during this time period), Dijkstra expressed his outrage about the Rice letter, suggesting that the attacks were *ad hominem* and not professional, particularly the final line of the letter, referring to the

---

[244] Gary Durbin, Oral history interview by Philip L. Frana, May 3, 2002, OH 368. Washington, DC. Charles Babbage Institute, University of Minnesota, Minneapolis.

[245] Edsger W. Dijkstra, "Letters to the Editor," 538.

poor innocent, novice programmers. However, his published response, in the same issue, was mild.[246]

Compromise solutions began to appear soon after Dijkstra's letter was published. Robert Fenichel's 1971 article, attempting to define programming languages that could index more correctly and free programmers to use go to statements within a clause that confirmed the indexing was in the correct position to be used, was one such common example.[247] These compromise solutions would address Dijkstra's specific complaint about call stacks (mathematical indices) without addressing the broader problem of the go to statement.

While the academic nature of most of the articles leaves the reader with the impression of a mild debate, the community of programming specialists felt divided. In B.M. Leavenworth's (contemporaneous) history, he explicitly calls it a controversy in his abstract, stating the paper was, "A Brief History of the GoTo Controversy (Retention or Deletion of the GoTo Statement) is Presented."[248] Harlan Mills, in his article, says that in circulating "'Go

---

[246] Edsger W. Dijkstra, private correspondence, 1968, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin, 10.

[247] Robert R. Fenichel, "On Implementation of Label, Variables," *Communications of the ACM* 14 no. 5 (1971): 349-350.

[248] Leavenworth, B.M., "Programming With(out) the GOTO", *Proceedings of the 1972 ACM Annual Conference, Vol. 2* (New York: ACM, 1975), 782 - 786

To Statement Considered Harmful'… Dijkstra set off a controversy that has rocked computer science."[249]

I think Mills' assessment is accurate. I argue that Dijkstra, both in his tone and in his attack on a language construct that simplified commercial programming, magnified an existing divide between programming specialists. The literature illustrates a clear divide. Dijkstra's supporters were primarily programming specialists pursuing a mathematical theory of computation and his detractors were programming specialists who were pursuing pragmatic solutions for commercial computing. Both sides of the debate drew out luminaries in the field to argue their position.

While the broader community interested in computing would assume that the debate over the use of go to statements was resolved in the 1970s, in 1987 Frank Rubin published a letter to the editor, "'GOTO Considered Harmful' Considered Harmful."[250] In this letter, Rubin asserts that programming without the go to statement was inefficient and costly to the programming industry. *Communications of the ACM* received enough responses to this letter to publish on the topic for five months and according to the editor, Peter Denning, these

---

[249] Harlan D. Mills, "How to Write Correct Programs and Know It," in *Proceedings of the International Conference on Reliable Software, 1975* (New York: ACM, 1975), 363-370.

[250] Frank Rubin, "'GOTO Considered Harmful' Considered Harmful," *Communications of the ACM* 30, no. 3 (1987): 195–196.

letters fell evenly on both sides of the debate. The editor of *Communications of the ACM* noted that the letter had generated more response by far than any other problem the journal addressed. I argue that this zeal arises from the perceived slight felt by the more pragmatic programming specialists, while the more esoteric programming specialists focusing on formal verification could not recognize Dijkstra's letter as a criticism.

The controversy over the go to statement was only the beginning of the programming "revolution" that consumed the community for the next decade. The shift towards structured programming began even prior to the go to statement controversy. In 1965 Dijkstra published a paper titled, "Programming Considered as a Human Activity." In this paper Dijkstra asserted that modularized, goto-less programs lead to programs that would be more elegant, drawing an analogy between the method of proof construction and program elegance. This paper was setting the stage for the later, structured programming methodology, but did not have the impact of the later "Go To Statement Considered Harmful"

Dijkstra was not alone in his interest in modular programming and mathematical proofs. A 1966 paper by Bohm and Jacopini, referenced later by Dijkstra, described a proof that illustrated that all

programs were made up of three control structures. In 1969, Naur published about the importance of mathematical proof of algorithms, as did Robert W. Floyd. In the background of these publications, Dijkstra's personal communication illustrates that he and Nicklaus Wirth were talking at length about mathematical proofs and program structure, influencing both Dijkstra's Structured Programming article and Wirth's 1966 article, co-authored with C.A.R. Hoare, "A Contribution to the Development of ALGOL."

In 1969 Dijkstra wrote "Notes on Structured Programming" and while it was not published in a monograph until 1972, the manuscript was widely circulated as part of Dijkstra's enormous private correspondence. Dijkstra was infamous for the regular stream of EWDs, unpublished notes on programming that he indexed and often distributed to his friends. Prior to the publication of "Structured Programming," the structured programming methodology had already been dissected, debated and extended by actors in the field through this private correspondence.

Structured programming became the dominant programming methodology of the 1980s. Dijkstra created the methodology as a result of his deep seated concerns about complexity and verification, but the methodology was accepted and evolved in a particular context,

one that stressed the need for decreasing complexity and increasing verifiability. To see how Dijkstra's beliefs and the particular context of the 1960s influenced the origins, acceptance, and evolution of structured programming, let us now explore the methodology itself.

What is Structured Programming?

Dijkstra proposed structured programming as a methodological tool kit to aid programmers in the design and implementation of software projects. In its original incarnation, Dijkstra defined structured programming as the use of testing aids (mathematical induction, abstraction and enumeration) and stepwise program composition.

What is StepWise Program Composition?

Stepwise program composition is the element of Dijkstra's work that became synonymous with structured programming. Stepwise program composition is a process of systematic sequencing which will allow the structure of computations to be reflected in the organization of the program. To carry out this process it is necessary to decompose each "step" the program makes into its sub-actions, deciding as little as possible with each distinct step.

Essentially, Dijkstra wanted programs to be made up of blocks of code, where each block did only one task. So, to use an example, if one wanted:

Y = (X+Z – A) * B

One would instead write three functions:

1. Function Add: L = X+Z
2. Function Subtract: M = L-A
3. Function Multiply: Y = M * B

And call them in order.

While this is a minor example, if you think about a broader system such as a flight reservation system, this example takes on more depth. If an agent was in the process of booking a flight, there are several functions necessary:

1. Search the flights that best match the criteria
2. Enter the customer's name and billing information
3. Put the customer on the preferred flight.

In Dijkstra's system, each of these would be separate, but more importantly, within each function, they would be broken down further. For example, the Search function may first need to ascertain what

flights go between the two cities and choose the one at the preferred time, before displaying it. So instead of a single piece of code that conducts the search for the flights, Dijkstra's system would be more modular. Modularity is foundational in both structured and object oriented programming. Given that structured programming and object oriented programming developed in parallel, this intersection is illustrative of the direction the field was moving towards.

Much of the code written before the adoption of Dijkstra's structured programming was more sequential, moving step by step from the first logical decision to the last, and then jumping around within the code when steps needed to be repeated. Below is a graphical representation of the flight search example we have been exploring. It is in the style of pre-structured programming programs.

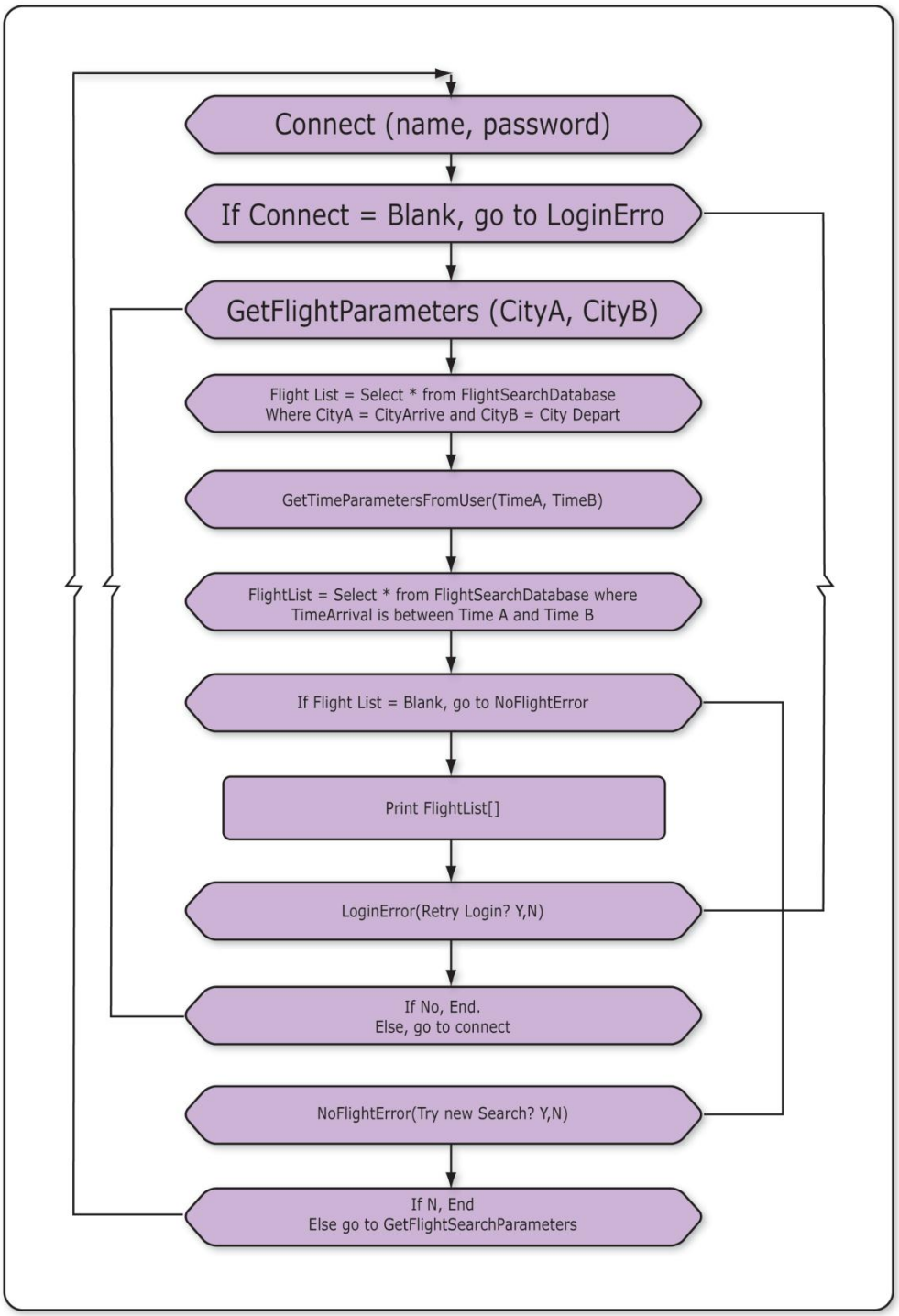


Figure 13: Graphical representation of an ad hoc program
This program would conduct a search for flights.

In contrast, a program designed with the structured programming principles can be more easily understood and evaluated. The diagram below illustrates the same program designed with structured programming principles. To see the detailed pseudo-code examples, see Appendix Two.

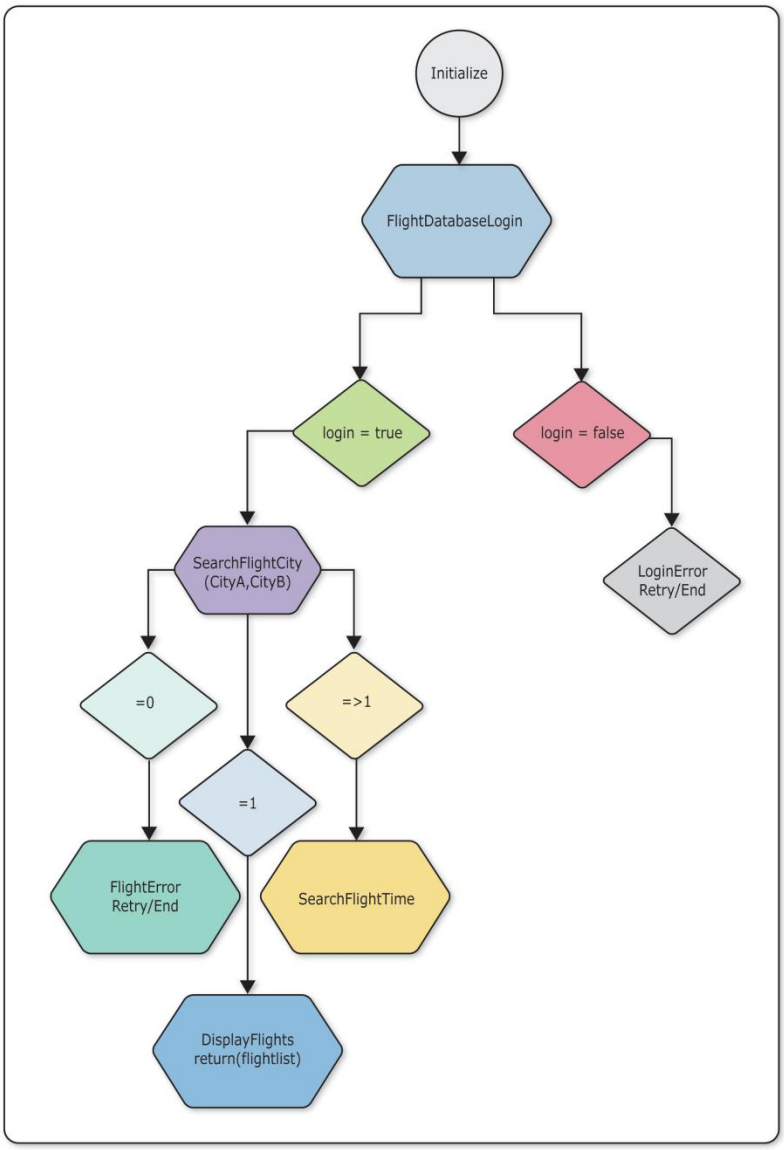


Figure 14: Graphical Representation of a Structured Program that Searches for Flights

The segmented code is better organized and easier to understand. Using Dijkstra's method, all of the outcomes are traced and accounted for, where in the less structured program the process jumps around at different points within the program. A flight reservation system is easy to conceptualize, which is why it is a useful example. However, if we think about applying this to a more complex function (we can imagine an actual flight control or auto pilot system) one can see how further reduction can simplify the problem.

What did Dijkstra Mean By Testing Aids?

Testing aids are exactly what they sound like: aids to help a programmer test that a program accurately completes the task in the way the programmer predicts. This clearly reflects the concerns of the programming community and was specifically targeted towards increasing the accuracy of programs and increasing the ability to verify that programs were accurate. Prior to Dijkstra's structured programming, the most common way of testing a program was path testing, where the programmer attempts to run the program with all possible variables. This is straightforward when programmers work on trivial problems, but as the programs get more sophisticated it becomes exponentially more complicated - just consider the number of all of the possible variables when a program flies a plane using an autopilot system.

Dijkstra describes three testing aids within his early paper that complemented path testing. The first is enumeration. Dijkstra defines enumeration as the "effort to verify a property of the computations that can be evoked by an enumerated set of statements performed in sequence."[251] He clarifies this through a series of examples that demonstrate that a program should be written in small segments, where each statement is analyzed to verify that it produces the correct output.

In an earlier, privately circulated paper titled "Toward Correct Programs," Dijkstra defined enumeration at length. To paraphrase Dijkstra's definition, he is asserting that each possible outcome of a statement should be traced, including the alternatives (for example, all outcome of an if-else clause are the outcomes of both the "if" and the "else" portion of the clause). While the concept was not new, the explicit statement of it was quite novel for programming specialists. [252]

The second aid that Dijkstra invokes is mathematical induction. Mathematical induction is generally defined as finding that a particular property holds true for all members of a sequence. This concept can ascertain the feasibility of a loop. If one proves that a statement holds

---

[251] Dijkstra, "Notes on Structured Programming," 7.

[252] Edsger W. Dijkstra, "Towards Correct Programs," EWD241. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin (approximate date: 1968).

true for a = 0, a = A and a = A + 1, you are proving that it will hold true for all values of a. Dijkstra then adds one final step to mathematical induction – that the loop exits when it reaches a particular state.

That a loop should be valid for all variables and should explicitly exit seems a straightforward explanation of proper programming practice, but it is important to realize that even in simple loops, to actually prove this statement mathematically can become quite complicated. This complexity substantially increases when loops become complex and nested. Dijkstra was arguing that to carry out this aid explicitly would help ensure the correctness of a loop.[253]

The final testing aid included in the "Notes on Structured Programming" is abstraction. Abstraction is the most amorphous of Dijkstra's concepts, as he himself admits. He describes it as pervading the entire field and uses the concept of an algorithm to convey the widespread nature of the concept. In Dijkstra's definition, abstraction promotes writing code that is not specific to any instance, but is instead a generalization that can support any specific instance. Generally, the use of abstraction in computer science allows for the re-

[253] Dijkstra, "Notes on Structured Programming," 8.

use of code by emphasizing what an operation does as opposed to how it works.[254]

Controversy

Structured programming was controversial from its inception. In 1970, prior to the publication of *Structured Programming*, while being privately circulated, Naur critiqued the methodology, foreshadowing some of the criticisms that would come from the community of programming specialists. Naur noted that structured programming wasn't realistic. In the unpublished document, "Notes on Structured Programming," Dijkstra used an example of stepwise program composition. Naur argued that this example was disingenuous, because Dijkstra couldn't escape his own assumptions.

Essentially, Naur believed that because Dijkstra already had an understanding of the goal of his program, he was unable to really implement stepwise program composition because he was always moving towards the result he was expecting. Naur was arguing that a programmer needed an understanding of the end goal of the program to use stepwise program composition. This criticism, that stepwise programming was unrealistic because programmers needed an understanding of the goal they were working towards, was a complaint

---

[254] Ibid., 11-39.

that numerous computer scientists made about the structured programming methodology.

Naur argued that, alternatively, in the initial phase, one should make a conscious attempt to collect all of the facts that impact the solution. Naur also suggested that one should explore several alternative approaches before deciding on the best approach, defining the best approach to be the one that encompasses all of the facets of the problem. He then argued that there was a better alternative to Dijkstra's example program. Naur is illustrating a more "top down" design approach, an approach that is later subsumed into the definition of structured programming.

Naur's other significant criticism was about the choices Dijkstra highlighted for the types of control structures. A control structure is simply a language construct that moves away from the main program body. The go to statement is part of a broader category of sequencing statements that instruct the program to do something, without any conditions. Selective control structures are constructs with the if/else structure, and Repetition control structures are loops that do a specific task while a condition holds true. Naur is arguing that he believed there was little justification for not including any sequencing control

structure.[255] This complaint was later taken up by the community of programming specialists.

Later critics, like Douglas Jones, echoed these sentiments, arguing that "the structured programmer often produces code in an order relatively unrelated to the order required by a specific programming language. The task of reordering the structured code to the specifications of the chosen language is tedious and error prone."[256]

During the controversy the structured programming methodology was often critiqued as impractical, but it was also poorly understood. One specific example is in a 1973 article by T.E. Hull: "Structured programming is a common topic of conversation these days. It appears to be an idea that has finally arrived, and it is 'in' to teach about structured programming, even if we don't quite understand exactly what is meant by it."[257]

This tone was echoed in Peter Denning's 1974 article, "Is it Not Time to Define 'Structured Programming'?" He argued in the

---

[255] Peter Naur, personal correspondence, 1970, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

[256] Douglas E. Jones, "A Programming Aid for Structured Programmers, in *Proceedings of the 1974 Annual ACM Conference Volume 2, January 1, 1974* (New York: ACM, 1974), 676.

[257] T.E. Hull, "Would You Believe Structured FORTRAN?" *ACM SIGNUM Newsletter* 8, no. 4 (1973): 13-16.

conclusion of the article, "If we are going to be so enthusiastic in our pursuit of happiness by the path of 'structured programming,' should we not at least preface our discussions of it with a definition? Should we not apply our (newfound) criteria of precision and clarity to our discussions of programming too?"[258] In a later article Denning also highlighted the craft versus science divide that structured programming incited in the field. Structured programmers often saw themselves as scientists; many early promoters were in academia and were interested in mathematical verification. Opponents of structured programming often saw programming as craft, with the creativity that term implies, or an industry, with a practicality that precluded the mathematically elegant aspects of Dijkstra's methodology. In this article Denning states, "Programming involves complex mental processes and a great deal of creativity and ingenuity. No one believes seriously that creativity is structured. To impose a structure on the process by which a programmer works is to limit his creativity."[259]

Stephen W. Smoliar continued to fan the flames of separation between academic and practical programmers in the Letter to the Editor, or Forum, section of *Communications of the ACM*. Smoliar

---

[258] Peter J. Denning, "Is it Not Time to Define 'Structured Programming'?" *ACM SIGOPS Operating Systems Review* 8, no. 1 (1974): 6-7.

[259] Peter J. Denning, "Two Misconceptions about Structured Programming," in *ACM 75: Proceedings of the 1975 Annual Conference, 1975* (New York: ACM, 1975), 214.

wrote that while both the ACM annual computer science conference and the ACM SIGCSE Symposium confirmed that structured programming was this year's trend, the community shouldn't get too carried away with a fad. Instead he argued for the importance of FORTRAN in the real world and further commented that FORTRAN could be adopted to use the benefits of structured programming.[260]

Moreover, Smoliar lashed out about Dijkstra ignoring the use of comments as a method of improving the readability of the program. This is a strange type of logic that suggests he read Dijkstra's methodology as exclusive, in the sense that anything not said explicitly was excluded from the methodology, where Dijkstra's intent was to be inclusive of any successful, compatible techniques that improved the quality of programs. This is reflected on page 83 of Dijkstra's manuscript, "Notes on Structured Programming." Dijkstra concluded his second example of stepwise refinement by saying that the process would lend programmers appreciation for elegant solutions, instead of one that was simply adequate.[261]

---

[260] S. Smoliar, "Letter to the Editor: On Structured Programming," *Communications of the ACM* 17, no. 5 (1974): 294.

[261] Dijkstra, "Notes on Structured Programming," 83.

Structured programming supporters did not sit idly by and absorb these criticisms. In response to the critique by Smoliar, David Gries wrote:

> Structured programming, then, is an approach to understanding the complete programming process. I fail to see why it has created a controversy. While it is true that some people go overboard on their belief in the preliminary results (don't ever use a goto, you must use only the conditional and while statements!), the majority of the people involved in sp [structured programming]are interested only in learning how to program better. Sp research will have an important and lasting impact on computing because it will lead to a better understanding of programming.[262]

Redefining Structured Programming

Structured programming was not a static phenomenon. It was developed more fully by the community and, at times, in contradiction to Dijkstra's wishes. This was the case with the most famous redefinition of Dijkstra's structured programming, a task undertaken by Harlan Mills.

Harlan Mills was a pioneer in programming theory. At the time structured programming was introduced Mills was working for IBM, but his work skewed heavily towards research and development. Moreover, despite Dijkstra's scathing critique, Mills was heavily

---

[262]David Gries, "On Structured Programming – A Reply to Smoliar," *Communications of the ACM* 17, no. 11 (1974): 655.

involved in the mathematical theory of computation. Much of Mills' work was centered on verification. However, unlike Dijkstra, Mills was focused on improving testing techniques. The root of Dijkstra's criticism stems from his desire to verify the design of programs to verify through mathematical proofs — formal verification. Mills used mathematics and high level statistics to improve the design of testing systems, attacking the problem of verification from a different angle. Both Dijkstra and Mills significantly contributed to the discipline, but Mills' work had a practical application that could be applied to improve existing programming. Dijkstra's eventual goal (mathematical proof of program correctness) is, today, still out of reach.

Mills clarified Dijkstra's original definition, describing how a Structured Program is to be written. Mills explicitly limited the number of control structures to be used within a program. He used only If Statements, Then Statements, the If-Then-Else Statement, and the While-Do Loop. He argues that this differs from simply "GoTo-free" programming, explaining that structured programs should be organized into "trees of program segments…each segment a prescribed size…and with entry only at the top and exit at the bottom

of each segment."[263] He calls this segmentation (what later became known as modularization) a "generalized data processing operation"[264] and labeled it the "Structure Theorem." Segmentation, or creating modules of code, is one of the most frequently cited solutions to the problem of complexity. When working in teams, if programs are segmented or modularized, programmers can work on individual segments, with little interaction between segments. Modularity has long been perceived as a method of decreasing complexity and improving programming as an industrial activity. As early as the work being conducted on the SAGE project, techniques were being used to create modular code.

Mills' second redefinition is a programming methodology based on the "Top Down" design of programs. Mills' Top Down methodology began with the "…integration code, at the system code, then subsystem levels and the functional modules…last."[265] Mills' Top Down approach emphasized planning and a comprehensive understanding of the system. This is distinctly different from Dijkstra's

---

[263] Harlan D. Mills, "Mathematical Foundations for Structured Programming," in *Tutorial on Structured Programming: Integrated Practices*, eds. V.R. Basili and F.T. Baker (Washington D.C.: IEEE Computer Society Press, 1981), 50.

[264] Mills, "Mathematical Foundations," 49.

[265] Ibid., 50.

stepwise refinement, where the smallest of decisions are made at each step.

Like Dijkstra's stepwise Program Composition, the stated benefits of Mills' Top Down methodology all relate to program correctness and as a solution to the complexity problem. Often in the literature, Top Down design is said to help improve the modularity of code and its accuracy. However, neither Mills nor his supporters go into detail to explain how this method of design helps in these areas. When reading the literature, there is little that suggests that Top Down design methodology would improve either modularity of the code or the accuracy of a program in any way more substantive than any other rigorous design methodology, including stepwise program composition.

Mills' third contribution is the "Correctness Theorem." Mills "Correctness Theorem" reiterates Dijkstra's plea for a formal mathematical system of verification.[266] In "How to Write Correct Programs and Know It," Mills recommends analyzing each functional specification in terms of a mathematical function, and using a

[266] Mills, "How to Write Correct Programs," 363-370.

flowcharting process to prove the program is correct, and then expanding on that specification.[267]

One of these flow charting processes was specifically geared towards structured programming. Developed by Isaac Nassi and Ben Shneiderman, one can see the specific avoidance of go-to statements and the focus on control structures. For example,

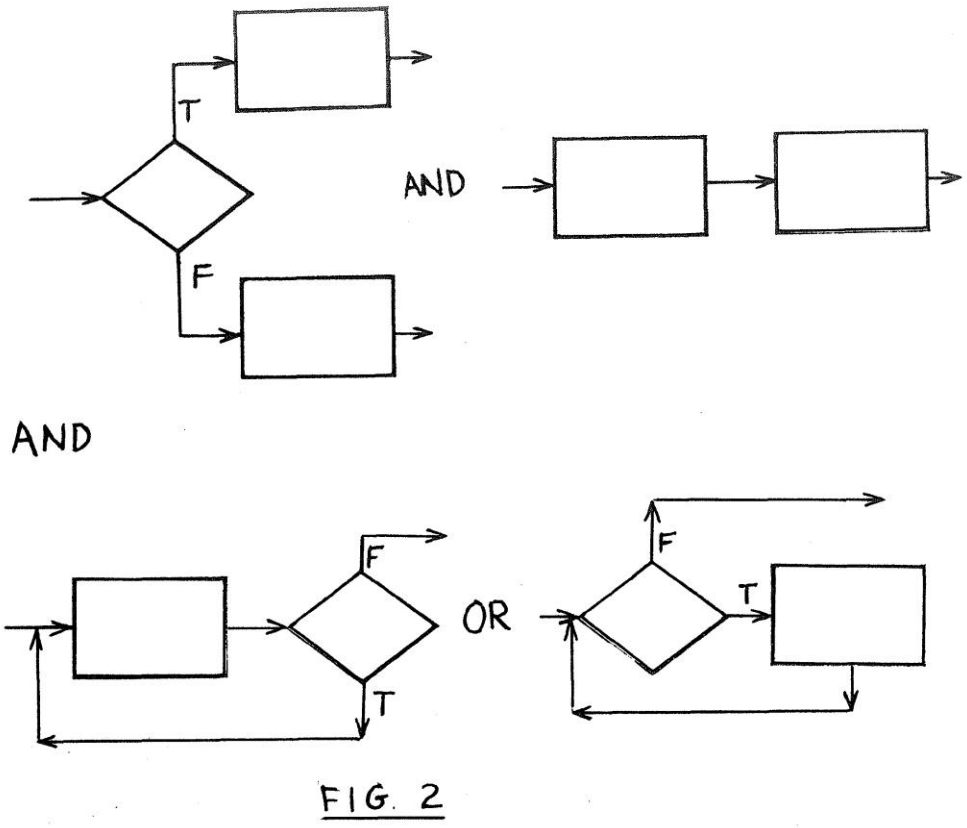


Figure 15

[267] This is the "Expansion Theorem,'" demonstrating how to expand out the limited control structures using the "Top Down" Corollary.

Source: http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/

Nassi Shneiderman Diagrams became the de facto standard for flow charting structured programs.[268]

Mills added formal commenting and indentation as a requirement of structured programming. Mills was not the only computer scientist advocating formal commenting and indenting systems to improve program readability. However, his position at IBM and influence in the community contributed to the acceptance of the importance of a formal system of commenting and indentation.

The most controversial aspect of Mills' attempted redefinition was the introduction of industrial organizational techniques, focusing on how to manage a team of programmers. Mills' work with T. T. Baker on the *New York Times* project dictated small teams, with three basic members – the chief programmer, a senior level programmer and a program librarian. The model was a centralized, autocratic structure. Goals and problem solutions were made at the highest level of authority. Tasks were well defined, complex and time dependent.

---

[268]Ben Shneiderman, *A short history of structured flowcharts (Nassi-Shneiderman Diagrams)*, May 27, 2003, available on the web at: http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/.

While programming specialists never seemed to accept this as a part of structured programming, management latched on to the concept.[269]

Mills' redefinition of structured programming was in some ways, at odds with Dijkstra's original conception of the methodology. In his top down design method he addressed the major criticism of stepwise program composition, which was that programmers needed to know their eventual goal when designing the program. He also addressed managerial concerns that Dijkstra had not considered. Dijkstra hated Mills' redefinition and popularization of his work. In one of his privately circulated notes he described IBM as "stealing the term 'Structured Programming' and under its auspices Harlan D. Mills trivialized the original concept to the abolishment of the goto statement."[270] This was simply not true. Moreover, Mills' correctness theorem had many of the vigorous elements that should have appealed to Dijkstra. In programming theory, Mills is credited with not only redefining Dijkstra's structured programming, but with popularizing the methodology. Not only was Mills seen as bringing the weight of IBM's approval to bear upon the programming methodology,

---

[269] Marilyn Mantei, "The Effect of Programming Team Structures on Programming Tasks," *Communications of the ACM* 24, no. 3 (1981): 106-113.

[270] Edsger W. Dijkstra, "What led to 'Notes on Structured Programming,'" EWD1308, Edsger W. Dijkstra Papers, 1948-2002, Archives for American Mathematics, Center for American History, The University of Texas at Austin.

he was also part of the first team to relate the story of a successful, large scale, commercial project, the *New York Times* online storage and retrieval system, completed using his redefined structured programming methodology.[271]

Oddly, though Mills has become synonymous with the first redefinition of structured programming, he was not the team leader of the project that is credited with providing the support for structured programming methodology. In actuality, Terry Baker was the team leader of the *New York Times* online storage and retrieval system. Anecdotally, it is said that the IBM programming team applied structured programming methods to the development of an indexing system for the *New York Times* research file. The project was completed efficiently and accurately and it was reported that managers of other companies were persuaded to adopt structured programming because of this success.[272]

There were several accepted redefinitions of structured programming in the years of debate following Dijkstra's introduction of the methodology. For example, Larry L. Constantine's 1974 paper, "Structured Design," redefined Mills' segmentation and Dijkstra's

---

[271] Donald MacKenzie, *Mechanizing Proof,* 57.

[272] James E. Tomayko, "Anecdotes," *IEEE Annals of the History of Computing* 12, no. 4 (1990): 269-276.

abstraction into a module, which contains one function and is able to be called as a standalone subroutine, where the internal workings of the function are hidden and the modules communicate via an interface.[273] Another refinement of the methodology was introduced by Victor R. Basili (the editor of several books on structured programming and an advocate of the methodology) and combined Mills' top down design with Dijkstra's system of stepwise refinement, by encouraging the continued decomposition of Constantine's modules.

By the early 1980s, consensus had formed around the meaning of the structured programming methodology, even if there was still debate over its efficacy. There were five accepted pillars of structured programming:

Structured Programs use a specific design methodology

Structured Programs are written using a combination of Top Down and Stepwise Refinement design principals. These principles are encouraged specifically because they help to produce code that is more easily verifiable and understood.

The resulting design must be modular & hierarchical

---

[273] Larry L. Constantine *et. al.,* "Structured Design," in *Tutorial on Structured Programming: Integrated Practices,* eds. V.R. Basili and F.T. Baker (Washington D.C.: IEEE Computer Society Press, 1981), 148-152.

Basili's concept of modules with a single function is a standard part of structured programming. A segmented design is implied in both Dijkstra and Mills' work, but Basili's conception was unambiguous. Hierarchical, modular design allows programs to grow in size, while providing a logical structure that can be error checked. This reflects the community of programming specialists' concerns with program correctness.

By the 1980s, when structured programming was part of undergraduate computing courses and textbooks, modularity had acquired a strict definition. For code to be modular it must first be segmented, with a single entry and exit point. The module must be self-contained. Again, this reflects the computing community's concern with accuracy, as it allows for complexity metrics and error checking to be completed more easily. It also creates a logical structure for the program, as each module contains a minimum of functions, as we can see in the Flight Reservation example from figure 14.

By the early 1980s modularity had been further refined to encompass information hiding. In this refinement the module is unable to access data other than the information passed to it when it is called, through parameters. Users of the module choose it based on

the function it performs without necessarily having to know how it operates. This is an interesting overlap, as this type of information hiding is one of the basic principles of object oriented programming, the methodology explored in the next chapter.

Abstraction

Structured programs and their modules are abstract. To illustrate: instead of solving a particular equation (for example: 2 * 9 + 4 = A) a Structured Program should be able to solve all variations of that equation using variables (WX + Y= A). If we were to trivialize the example, creating two modules, each module should be able to solve its particular part of the equation (example: 2*9 = a), but should also be general enough to be used in other equations (example: WX = a).

However, by the early 1980s, the reason given for the necessity of such abstraction in structured programming is no longer Dijkstra's desire for accuracy, but for the reuse of code. It was most often applied to the reuse of code for the implementation of libraries. Dijkstra's desire for accuracy has been supplanted by efficiency and simplicity of management. As we will see in the next chapter, the sophisticated use abstractions, like modules, become a part of the object oriented programming methodology.

Limited control structures

A control structure is simply a language construct that moves away from the main program body. A go to statement is a control structure, hence, any definition of structured programming that does not undermine Dijkstra's intentions limits the control structures by excluding the use of go to statements.

Formal systems of documentation and indentation

The structured programming methodology includes proper documentation and indentation. The concept is not limited to a structured programming methodology, but proper documentation is particularly important when attempting to use modules as black-boxed code, where programmers use the function of the module without knowing its internal operations.

Results of the Structured Programming Conflict

It should be obvious that the problems of verification and complexity were not solved by the "Structured Programming Revolution." Dijkstra himself demonstrated this late in life when he wrote a letter of recommendation for Robert Boyer and J. Strother Moore, arguing that the pair distinguished themselves through their work on program verification. Dijkstra emphasized the importance of their work because the solution to the problem of verification was crucial to the viability of the computer industry as a whole and for

society to reap the benefits of the computing revolution.[274] In one of his final privately circulated notes, Dijkstra continued to lament both the problem of program correctness and complexity. He felt that the root of the controversy was that the prevention of accuracy problems, using formal verification techniques, was still better than the cure, using testing techniques:

> The moral is clear: prevention is better than a cure, in particular if the illness is un-mastered complexity, for which no cure exists…[275]

There was more work to be done after the almost universal adoption of structured programming methods.

In the late 1980s structured programming began to fall out of favor. More interest sprang up in the newly popularized object oriented programming techniques. The next chapter will illustrate how this newly dominant methodology also aimed to solve the problems of complexity and verification. Interestingly, both object oriented programming methodology and structured programming methodology were conceived in the late 1960s and early 1970s, but while structured programming became a topic of heated controversy, object oriented

---

[274] Edsger W. Dijkstra, Private Correspondence, 1981, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

[270] Edsger W. Dijkstra, "Forty years of Computer Science Devoted to Education," privately circulated note, 1989 EWD1051-1, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

programming was developed and revised with little fanfare. Moreover, in a letter to Wirth (as the editor of the *ACM*) in August of 1967, Dijkstra justifies his rejection of an article for publication in the *ACM* because the authors "miss[ed] a reference to Simula (Ole-Johan Dahl and … Kirsten Nygaard)." Dahl eventually coauthored "Structured Programming," publishing Dahl and Nygaard's concepts that are strikingly similar to object oriented programming.

Dahl and Nygaard created the Simula programming language, which is often described as a precursor to object oriented programming. The traditional narrative in programming theory describes Simula as containing the fundamental principles of object oriented programming methodology. Chapter six will explore the relationship between structured programming and object oriented methodologies and the role of verification and complexity in the fall of structured programming and the rise of object orientation. In my narrative I argue that this is not the case, that the true precursor to object oriented programming is Simula 67, the language Dahl and Nygaard created to solve shortcomings in the original Simula implementation.

Structured programming evolved from an abstract, mathematically rigorous style of programming to a strict and formal

method of design that then had to be translated into code. Despite claims of a "structured programming revolution," I argue that it was not a radical shift in programming, but instead a shift in the design of programs.

That structured programming was a shift in design, not programming, is supported by Loren Meissner stating at the 1974 ACM Annual conference that, "It has not been strongly enough emphasized that the structure of a program is pretty firmly established during the program design phase, and therefore that not much can be done during the coding phase to change or to augment program structure."[276]

Structured programming was, essentially, a methodological dead end. It was the consolidation and popularization of techniques that should have already been implemented in practice. More interesting than the actual methodology are the circumstances that contributed to its origins and acceptance. Dijkstra's stimuli in the creation of structured programming were complexity and verification. Programming specialists were attentive to these problems, creating a climate of acceptance for the methodology. However, as we shall see

---

[276] Loren P. Meissner, "A Method to Expose the Hidden Structure of FORTRAN Programs, in *Proceedings of the 1974 Annual ACM Conference Volume 1, January 1, 1974* (New York: ACM, 1974), 193.

in the following chapter, these problems drove the community's eventual acceptance of object oriented programming. Object oriented programming was developed in parallel with structured programming, but without the charisma and fanfare created by Dijkstra's personality, the methodology was essentially overlooked until the 1980s.

Chapter 5: Object Oriented Programming

The historians of science and technology often talk about the confluence of ideas. The Industrial Revolution and the Scientific Revolution can both be viewed as a junction in time when a number of ideas from different disciplines came together to form a sum that was larger than their respective parts. Physics in the early 20th century and computing in the mid-20th century are also examples of times when disciplinary boundaries expanded and ideas from different fields came together to significantly change the discipline. Object oriented programming, too, arose from a confluence of ideas, in this instance to create a new programming methodology.

Many programming specialists argue that object oriented programming is a radical change in how the discipline perceives and conceptualizes software. To explore this claim, and to situate object oriented programming in the broader history of software, I am going to begin by briefly explaining how object oriented programming differs from the conceptual understanding of software that underpins automatic coding systems and structured programming methodologies.

What is Object Oriented Programming?

Object oriented programming is a methodology that is organized around objects. This differs from traditional software design methods where the program is organized around processes.[277] The clearest example of this is in simulation modeling. For example, in a highway simulation program using traditional methods, the program could start with a time statement, placing events, like car accidents or exits, at random times over the run-time of the program. In an object oriented design the program would allow for the instantiation of many objects. So, instead of writing the program from an archetypal driver's point of view (the control program), which flows through the simulation effected by the previously inserted accidents or exits, there might be many instances of cars involved in incidents occurring at the same time. The behavior of cars would include accidents and exits, forcing the different instances of the cars to interact in ways that are far less predictable than a simulation with pre-existing incidents scripted into the simulation.

The differences between a structured program and an object oriented program are clearly visible in diagrams depicting program design. The diagrams below illustrate the use of a heating system.

---

[277] L.F. Capretz, "A Brief History of the Object-Oriented Approach," *ACM SIGSOFT Software Engineering Notes*, 28 no. 2 (2003): 1-10.

The first diagram depicts a structured programming implementation. This is clear because the program design is process oriented. Processes like “Test Status” and “Change Temp” are highlighted, while the buttons themselves are part of the flow processes.

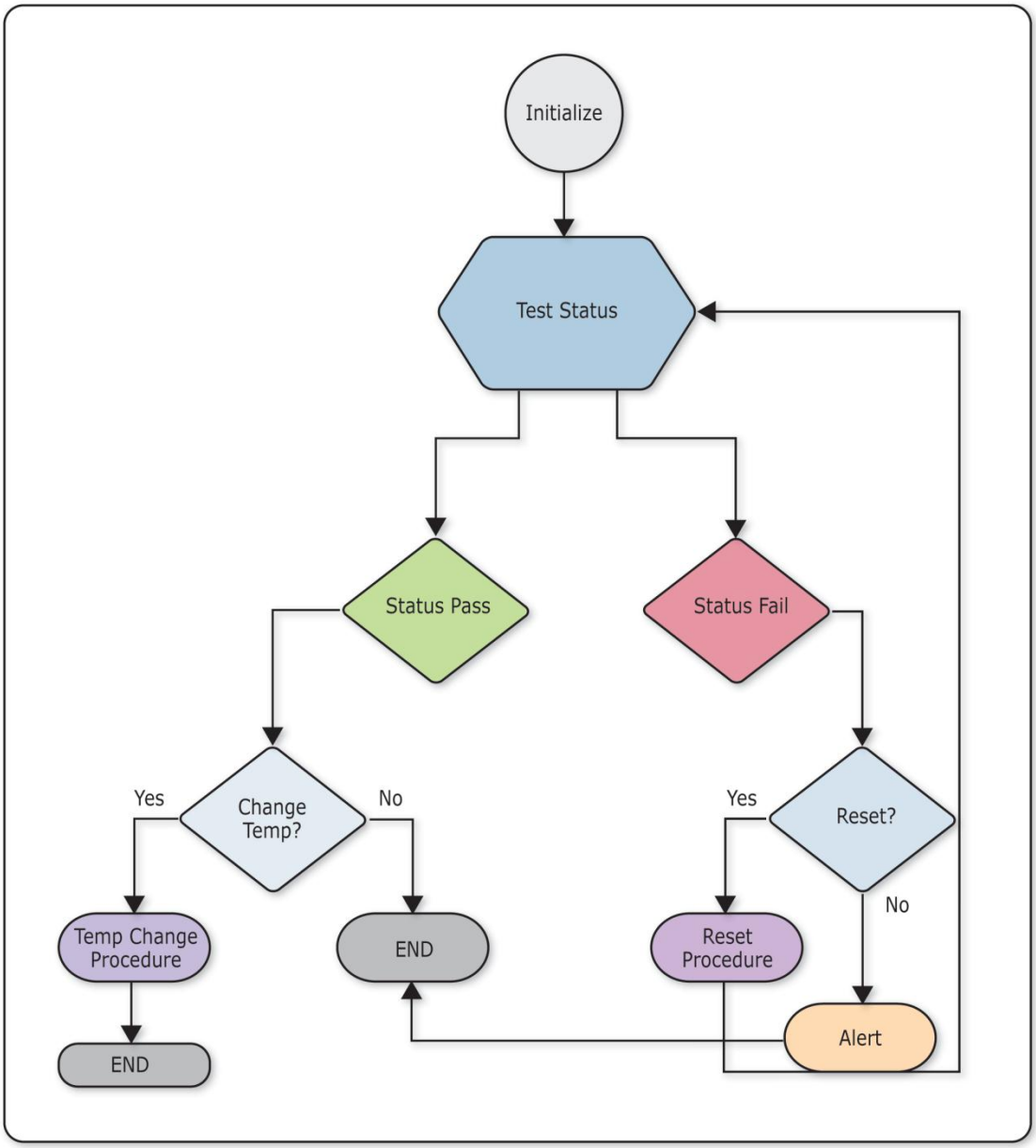


Figure 16: Graphical Representation of a Structured Program to Control a Heating System

Compare this to the following object oriented description of a heating system. "Timer" and "Furnace Status Indicator" are highlighted as classes of objects, which the Operator Interface Class controls:

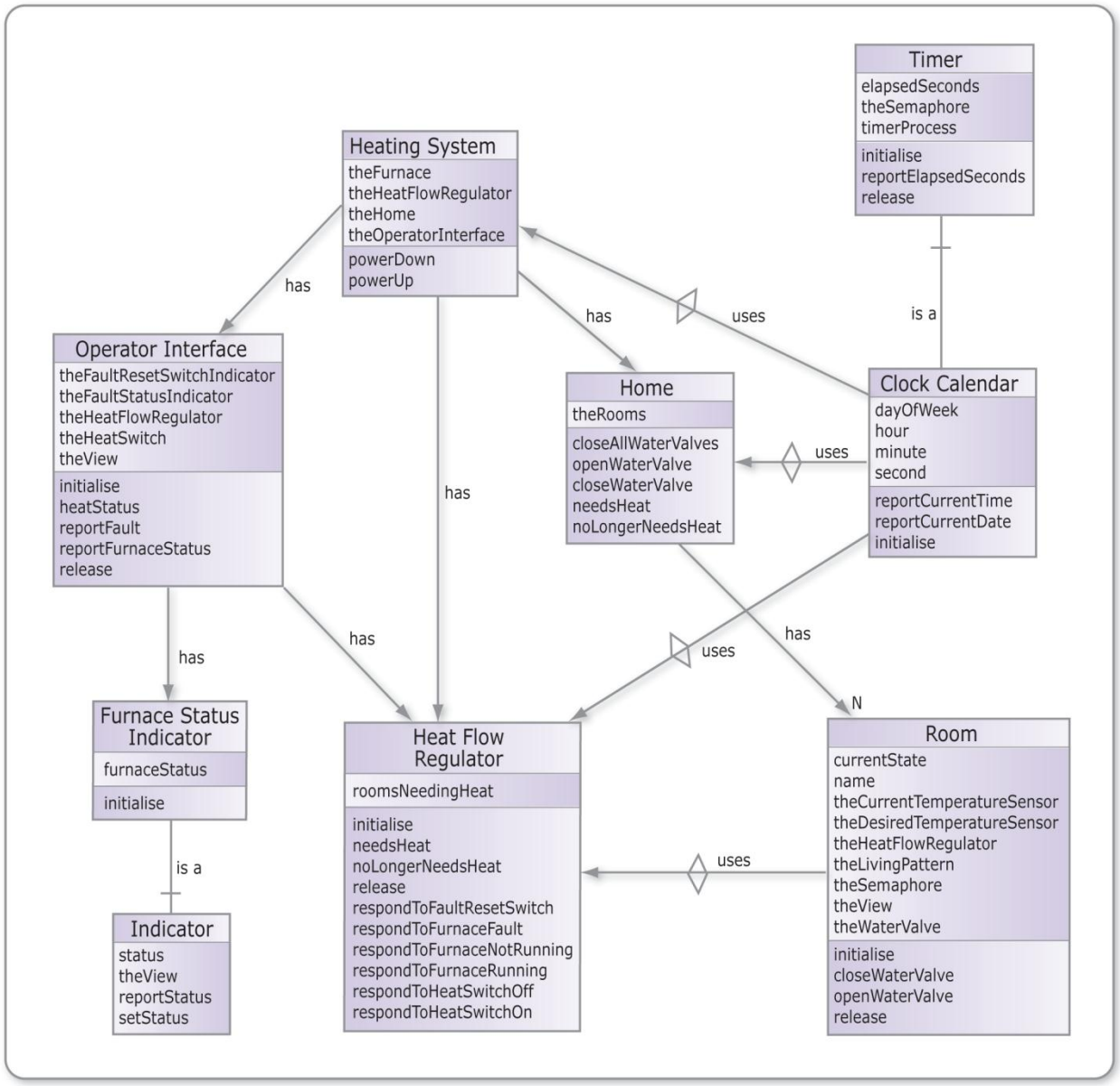


Figure 17: Graphical Representation of an Object Oriented Program to Control a Heating System

These diagrams illustrate that there is a distinct difference between object oriented programming and previous programming methodologies, particularly structured programming. I argue that this change centered on the world view of the programmer. In object oriented programming the focus is on objects communicating messages between each other, while in previous methodologies the goal was to accurately simulate the logical processes, in a step-by-step fashion. In the object oriented programming methodology the primacy of objects over processes is a shift of perspective that changes the way programmers view software.

Object oriented programming is a shift in perspective, requiring an object-centric world view, but it is more than that. The computing community expanded the definition of object oriented programming. The current conception of object oriented programming requires the language to support not only an object oriented worldview, but also a number of features. Many of these features were long held tenants of good programming practice, but in object oriented programming languages these features were then enforced by the language.

Object oriented programming languages support class structures, inheritance, encapsulation, abstraction of data,

polymorphism, and recursion.[278] I will define these features in detail and explain how they differ (or do not differ) from earlier programming methodologies. The evolution of the object oriented programming methodology demanded not only new programming languages, but also new programming environments that provide a broad range of tools and libraries that support (and at times, enforce) these features.

Classes

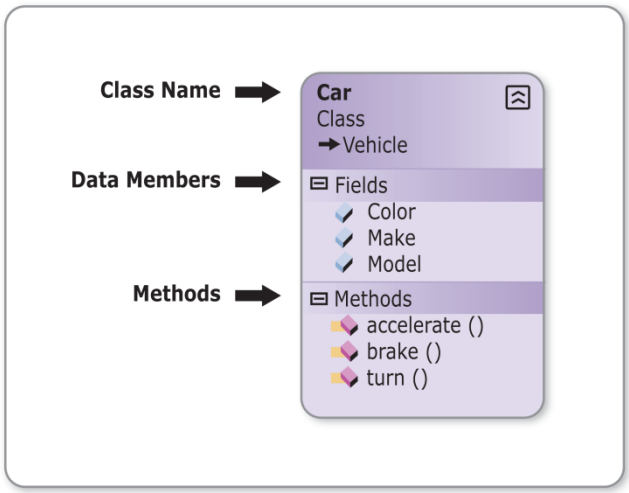


Figure 18: Diagram of a Class

Classes define the abstract characteristics of an object. These characteristics are its attributes and its behaviors. Characteristics are usually called fields, while behaviors are methods or functions. For example, the Class "Car" would include all of the fields (or members)

[278] Deborah J. Armstrong, "The Quarks of Object-Oriented Development," *Communications of the ACM* 49, no. 2 (2006): 123-128.

and methods (or functions) of a "Car" – color, make and model (fields) and accelerate, brake and turn (methods).

Objects are then particular instances of classes. To return to our example – the class "Car" could instantiate an object "Your Car," which happens to be a red Chrysler Sebring. This particular set of attributes is then defined as the state of the object.

The methods of an object are that object's behaviors. Returning to our previous example, cars can accelerate, brake and turn. Accelerate() would then be one of the methods of "Your Car". However, the car's behavior isn't dictated by the car itself. It is instead commanded by a driver. Therefore, there may be an additional Class "Driver," which instantiates the object "Your Driver". The "Your Driver" object would then tell the object "Your Car" to turn, by invoking its method, turn().[279]

The class structure may seem similar to modularized data in structured programming. However, the class concept is integral to the object oriented world view. Objects are an instance of the class. In structured programming, these modules, instead, contain processes.

---

[279] B. Stroustrup, *The C++ Programming Language* (Murray Hill: AT&T Labs, 2000), 732 – 733.

Inheritance

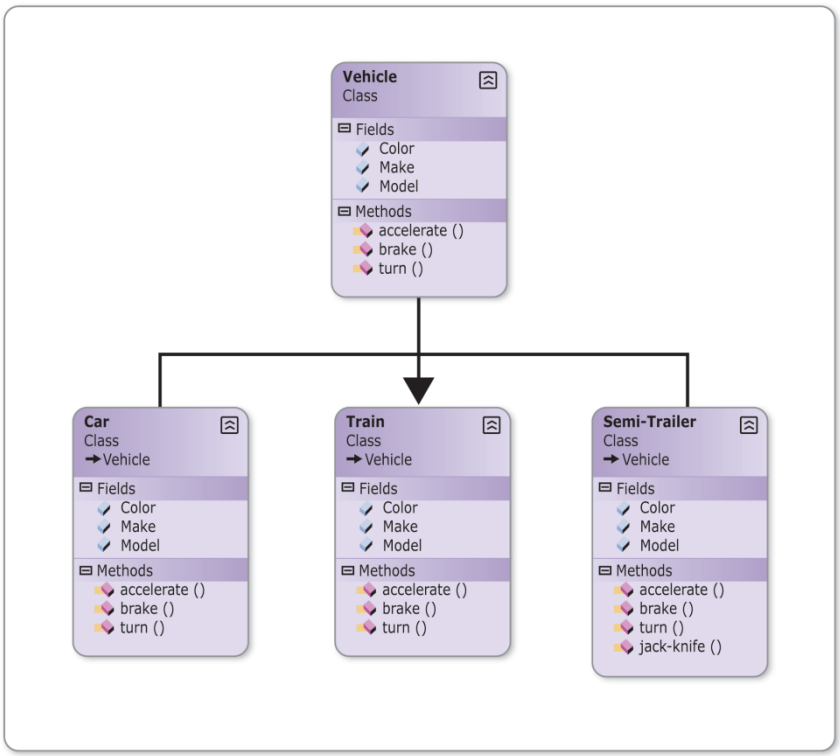


Figure 19: Diagram Representing Inheritance

Inheritance is an 'is-a' relationship: "Your Car" is a "Car." A "Car" is a "Vehicle." "Your Car" inherits the methods and traits of both "Car" and "Vehicle." Inheritance invokes the concept of "sub-classes", which are more specialized versions of a class and "inherit" all of the attributes and behaviors from the parent class. For example, the class "Car" might actually be a sub-class of a broader class "Vehicle," which has the sub-classes "Car," "Train" and "Semi-Trailer." The methods, accelerate(), brake() and turn() might all be inherited from the parent class "Vehicle." Sub-classes can also introduce their own attributes

and behaviors. For instance, "Semi-Trailer" might include the method jack-knife().

This is an important feature of object oriented programming because for all of the sub-classes of car, the inherited methods (accelerate(), brake() and turn()) and traits (color, make and model) need only be written once — even if the sub-class needs to alter a trait or method. For example if "Vehicle" had a trait that set the number of tires to four, the "Semi-Trailer" sub-class would need to alter that trait, but it wouldn't affect the other types of vehicles, like cars or trucks.[280]

Inheritance is a direct result of the object oriented world view. Shifting the primacy from processes to objects presents this 'is-a' relationship, where a train can inherit the characteristics of a vehicle. When working with processes, this relationship is unclear.

Encapsulation

Encapsulation is a form of data hiding, where the data and methods are stored within the object. This conceals the functional details of the class from objects that use it. While a "Driver" class might need to use the method, turn(), from "Car," it doesn't need to know how the method turn() works.

---

[280] Ibid., 38-39.

One important benefit of Encapsulation is that while the interface must stay the same, the way the method works is completely separate, allowing the programmer to change the underlying code. This helps with production schedules - if one programmer, programmer A, is working on the class "Driver" they can use the defined functions from "Car" without waiting for a second programmer, programmer B, to complete the sub-class "Car." It also helps with maintenance. If a more efficient way of coding the method turn() is found 3 years after the software is completed a third programmer, programmer C, is able to make changes to it without being concerned that they will be making inadvertent changes to the objects that use the methods of "Vehicle."[281]

Encapsulation is an extension of data hiding, a concept originating with modular code in the early 1970s. This is one of the benefits of the slow evolution of object oriented programming. The methodology was fluid, acquiring the most useful programming innovations and assimilating them over 20 years.

Abstraction

The basic idea of abstraction in computer science is based on the idea of generalization — a concept or idea not associated with any

[281] Armstrong, "The Quarks of Object-Oriented Development," 123–128.

specific instance. However, as computer science has grown abstraction has become a sophisticated concept, applied to a number of different areas.

Abstraction can be applied to the control of the flow of the program. When talking about control abstraction, programmers are referring to the most basic use of programming languages. Everything from Grace Hopper's A0 compiler to the most complex languages currently used, use control abstraction. Programming languages allow the programmer to avoid having to physically manipulate the machine language contents of registers to achieve their desired outcome by using symbolic notation.

Data structures are also abstractions. The stack structure we saw previously in this dissertation is an abstract idea. That stack concept can be implemented in either the hardware or the software. In object oriented programming, classes themselves are an abstract data structure. Classes are a grouping of metadata, which represents some real world object.

Abstract data structures were not new to programming theory when they were introduced in the object oriented programming worldview. Abstract data structures were a part of the 1960s discussion about verification and complexity. Abstract data structures

were included in the definition of structured programming in the form of modules. Object oriented programming extended this, basing the entire worldview on classes of objects - abstract data structures.

Polymorphism

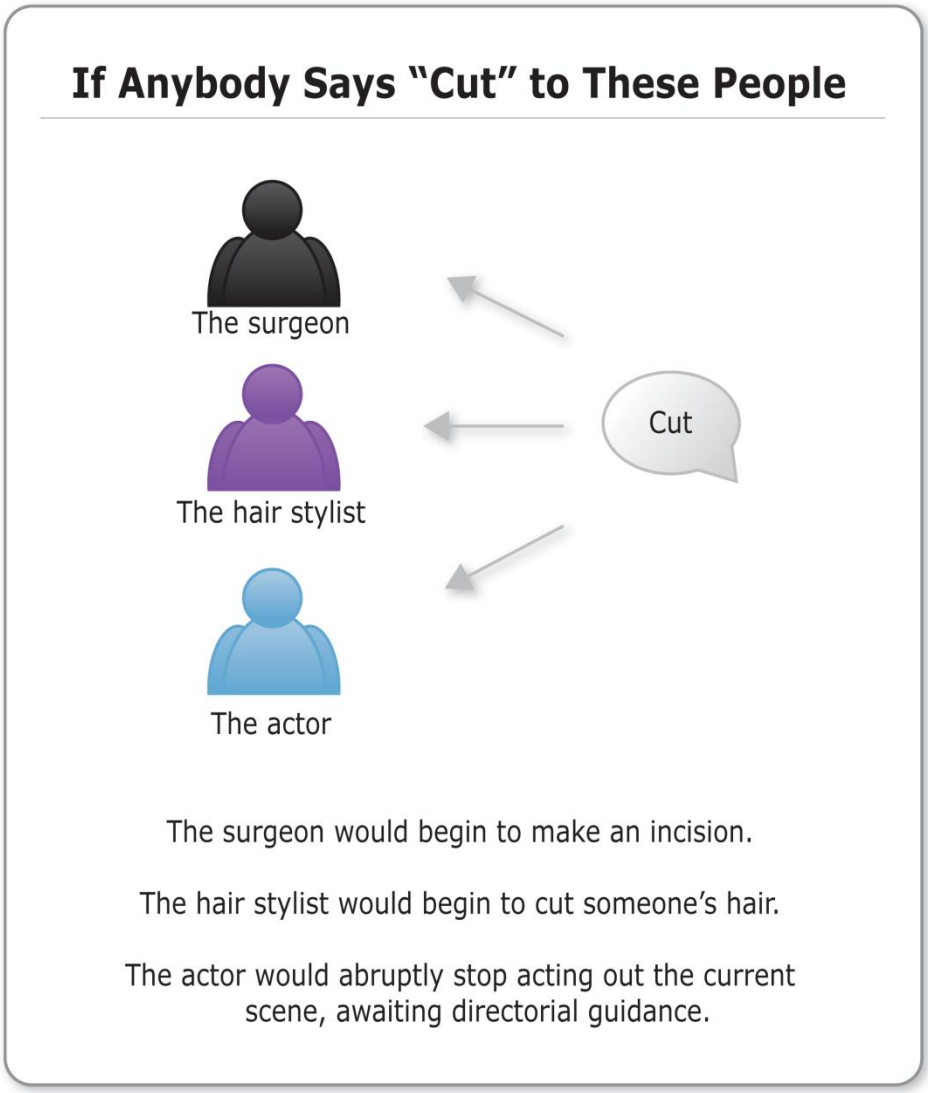


Figure 20: Graphical Representation of Polymorphism

Polymorphism is the ability of different objects to respond to functions of the same name, while yielding different results. So, for example, if you had a Class "Animal", with the function speak(), an object of sub-class "Dog" will bark, while an object of sub-class "Cat" will meow because the function, speak(), is redefined in the sub-classes. This is what is called over-riding polymorphism.

Overloading polymorphism is the use of one method or operator to perform different functions depending on the implementation. One well known example of this is the "+ operator," which is used to perform integer addition, float addition, and list concatenation.[282]

Recursion

Figure 21: Graphical representation of recursion

[282] Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys* 17, no. 4 (1985).

Recursion is when a method or function is defined in terms of itself. A classic example of a recursive procedure is the function used to calculate the factorial of an integer.

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

So for example, in C++ the following function would calculate factorials. Note the use of the function fact() within the function, fact().[283]

```
int fact(int n)
{
    if (n == 1)
return 1;
    else
return n * fact(n - 1);
}
```

What are the Origins of Object Oriented Programming?

Alan Kay coined the term "object oriented programming" during his work on Smalltalk at Xerox PARC, but the history of the object oriented methodology begins earlier than the Xerox PARC work and extends well past it – beginning in the 1960s and spanning the 1980s through to current programming practice. I argue that the first object oriented language was Simula 67. This differs from the traditional

[283] Armstrong, "The Quarks of Object-Oriented Development," 123–128.

account of programming theory, which identifies Simula as the first object oriented language.

Simula 67 was an extension of the Simula programming language created by Ole-Johan Dahl and Kristen Nygaard in 1963, at the Norwegian Computing Center. Dahl and Nygaard elaborated on Simula in the late 1960s, creating the Simula 67 language.[284] The traditional storyline points to three elements of the original 1963 Simula implementation as the origins of object oriented programming. I will briefly describe them:

Simula focused on "permanently present active components" and a "variety of transient passive components" as a way of modeling a dynamic system. These are essentially modules that dealt with functions (permanently present, active components) and data (transient, passive components). In the traditional narrative, these "transient passive components" can be seen to act as abstract data structures, the building blocks of object oriented programming.[285]

Simula used the ALGOL stack as the method of execution as a form of object orientation. Essentially, Simula would use the ALGOL

---

[284] Kristen Nygaard and Ole-Johan Dahl, "The Development of the Simula Languages," *SIGPLAN Notices* 13, no. 8 (1978).

[285] Kristen Nygaard and Ole-Johan Dahl, "Simula: An ALGOL-Based Simulation Language," *Communications of the ACM* 9, no. 9 (1966).

stack as the Main function, the function that would manipulate the abstract data structures (classes) written in Simula. So, if one needed to simulate traffic lights, Simula provided a data structure that could contain the functions of the traffic light and then use the ALGOL stack to call those functions. While this structure supports object oriented programming, it does not require the program to be object oriented. It is instead a form of modularization.

The third element used in the traditional story to support Simula as the first object oriented programming language is a statement made by Dahl and Nygaard. In the Simula language specification, Dahl and Nygaard made a specific claim that the language would be problem-oriented, not computer oriented. This claim was later adopted as one of the key concepts in object oriented programming.[286] However, I argue that when applying it to Simula, the statement was rhetorical.

I argue that this is rhetorical because in the original Simula implementation these concepts were at most ill-defined and the terms used contradictory. In contemporaneous literature Dahl and Nygaard refer frequently to a process-based method of explanation. In hindsight, after starting work on Simula 67, Dahl described Simula as

---

[286] Ibid.

regarding the world as a collection of programs that interact and exist in parallel, very different from the class concept that they used in the Simula 67 language description.[287]

Unlike computer scientists, I argue that while Simula may have had concepts that, when developed, contributed to the object oriented framework, the original Simula language did not conceive of objects. Like many early simulation based languages, the need for an abstract data structure was realized, but this wasn't described in terms of objects or classes.[288]

Dahl and Nygaard quickly found elements of Simula that they wanted to improve upon. Soon after Simula's final definition was released and UNIVAC began distributing the Simula language, Dahl and Nygaard began working on a general purpose programming language that was later named Simula 67. Heavily influenced by Hoare's 1965 work on record handling, Dahl and Nygaard redefined

---

[287] Ole-Johan Dahl et al., "Some Features of the Simula 67 Language," in *Winter Simulation Conference: Proceedings of the Second Conference on Applications of Simulations, December 1968,* (New York: Winter Simulation Conference, 1968).

[288] LISP, for example, had an abstract data structure, but this wasn't conceived of in terms of object or classes. Guy L. Steele, Jr. and Richard P Gabriel, "The Evolution of LISP," in Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. eds., *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993* (New York: ACM, 1994).

Simula 67 to be based on the concept of classes of objects and introduced inheritance using Hoare's subclass concept.[289]

The programming modules defined in Simula 67 were based not on the processes of Simula, but on objects. Further, Simula 67 had a novel way of presenting these objects so that each object stores its own behavior and data. Classes and inheritance are key concepts that define object orientation, but they also result in a style of modularity that appealed to Dijkstra as part of his structured programming paradigm.[290] Interestingly, while Simula 67 is, in retrospect, the first object oriented language, it was never called object oriented. The term itself wasn't used until Alan Kay's work on Smalltalk in the mid-1970s. Simula 67 was, instead, originally perceived to be a language that supported structured programming.

Despite Simula 67 being the first implementation of object oriented programming, it had very little impact on programming theory. A document search demonstrates that there was little interest or controversy over Simula 67's introduction, when plotted against the

---

[289] O. J. Nygaard and K. Dahl, "The Development of the Simula Languages," in *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993* eds. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (New York: ACM, 1994).

[290] Nygaard and Dahl, "The Development of the Simula Languages," 439.

number of articles that were written on FORTRAN during the same time period:

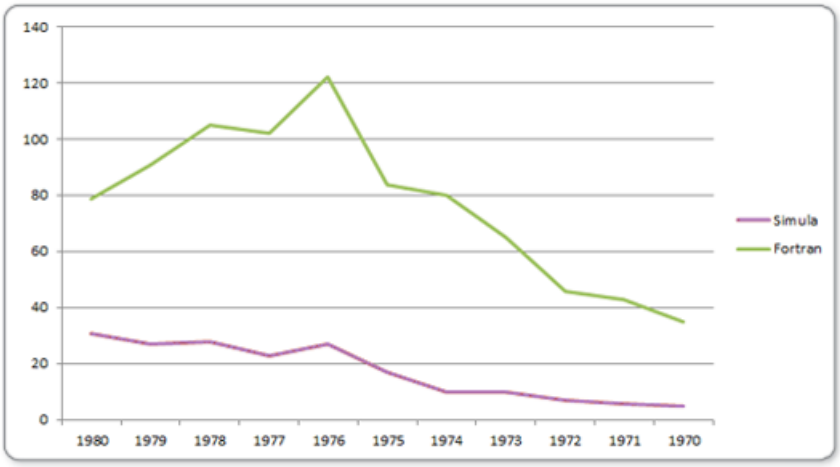


Figure 22: Comparison of the number of ACM publications on Simula and FORTRAN between 1967 and 1980

Also illuminating is the number of Simula articles versus the number of structured programming articles published between 1967 and 1980:

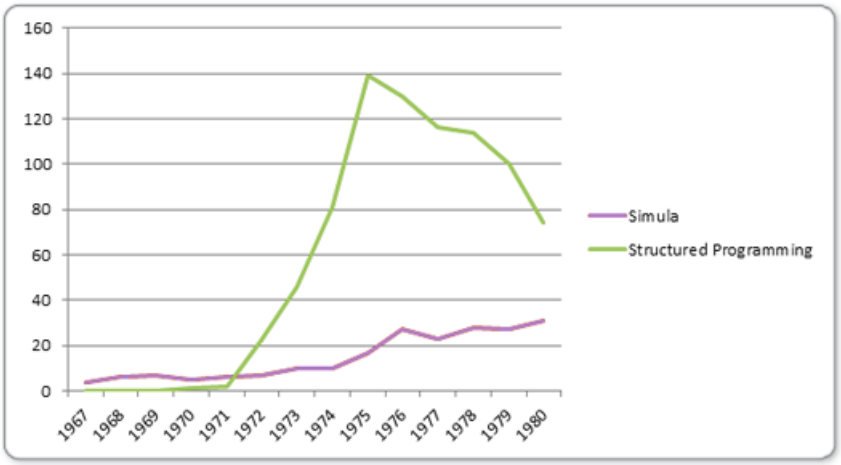


Figure 23: Comparison of the number of ACM publications
on Simula (both Simula AND Simula 67 included) and Structured Programming between 1970 and 1980

This chart illustrates that the impact of Simula and the later Simula 67 was negligible when compared to the controversy of the structured programming debate that dominated the 1970s. Historically, this is interesting because the concepts contained in Simula 67 became the dominant paradigm in the 1980s.

Simula 67 may not have made a distinct impact on programming theory, but the concepts did make quite an impact on Alan Kay as the creator of Smalltalk. Kay was a prolific contributor to programming theory and computing, more broadly. Mentioning Kay's contributions does not serve only to pay homage to a great contributor to the

computing community; his view of Smalltalk is colored by these contributions. Kay's early focus in computing was on its dissemination to a broader pool of end users. During his time at Xerox PARC his focus was on personal computing. Kay was inspired by Ivan Sutherland's work on the Sketchpad (described below). Kay wanted to create the Dynabook – a laptop-style computing device with a stylus sensitive screen, mouse and keyboard as input, which would run something like Sketchpad. Because of this focus, for Kay Smalltalk is still primarily a programming language that supports the goal of personal computing.[291]

Smalltalk was the first fully fledged implementation of an object oriented programming environment. At the Second History of Programming Languages Conference, Kay described LISP and Simula 67 as the major influences on Smalltalk.[292] LISP is not object oriented, but it is reflective and recursive in a peculiar way - LISP was written in LISP. What I mean by this is that the LISP Eval (Evaluation) function was implemented in machine code and the Eval function was then used as the interpreter for the entire language. The Eval function in LISP evaluates the arguments, applies the function to the

---

[291] Alan C. Kay, "The Early History of Smalltalk," in *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23,* 1993, eds. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (New York: ACM, 1994).

[292] Ibid.

arguments, and returns a result. LISP compilers or interpreters make use of this function to parse LISP applications.

LISP also has an interchangeability of data and functions. The primary representation of the source code is a list structure; this list structure is also used for the main data structures in LISP. As a result, the run time is significantly improved and the programmer can easily ascertain problems using the interpreter. Moreover, LISP functions can be manipulated and created within the source code for a LISP application.

In this way LISP is what Kay called a crystallizing language. By this, Kay meant that the language crystallizes a particular idea, in this case the idea of recursive, reflective programming. Smalltalk was similarly recursive and reflective, and the benefits of this style of programming contributed to Smalltalk's popularity.[293]

The influence of Simula 67 is more direct. Kay used the object oriented world view and the class concept that were introduced in Simula 67. In Kay's view the importance of object orientation was not only in the use of abstract data structures, but in the communication style of an entirely object oriented environment.

---

[293] John McCarthy, "History of LISP," *SIGPLAN Notices* 13, no. 8 (1978).

In this environment, objects would send messages to each other, where each object was essentially a tiny computer – with all the abilities of the computer encapsulated within it.[294] This reflected Kay's world view of personal, networked computing, bringing his macro and micro ideas into the same domain. While this world view, affording primacy to this style of messaging, is rarely articulated as the explanation for the popularity of object oriented programming, we can see that networked personal computing is a macro reflection of the type of communication performed between objects in object oriented programming. Figure 16, used for illustrating an object oriented system, could just as easily be turned into a representation of a network.

Kay did not just offer a name for the object oriented methods being used in Smalltalk. He also contributed to the extension of object oriented programming by including the idea of encapsulation. Encapsulation is a form of data hiding, and data and methods are stored within the object concealing the functional details of the class from objects that use it. So, to return to the class "Car" used in earlier examples, the interface of the object provides the programmer with the understanding that they can use the accelerate method, but the

[294] Kay, "The Early History of Smalltalk."

programmer is shielded from the details of how the method accelerate works, they just know that calling the accelerate method effects the object Car in a specific way. Smalltalk was not only a programming language. It was a fully-fledged programming environment which supported, among other features, data hiding, resulting in encapsulated objects being the building blocks of the programming language. This is what Kay is referring to when he talks about Smalltalk being a unification of the object oriented methodology and the principle of extensibility.[295]

Kay's use of the ideas that appeared in LISP, Simula 67 and Sketchpad guided the design of Smalltalk, as did the applications Kay wanted users to write in Smalltalk. Kay conceived of Smalltalk as a language for a personal machine, which would allow end users to create tools like simple databases and drawing programs in a programming language. Because of Kay's focus on simplifying the language for end users, Smalltalk served to overcome several of the problems of complexity that plagued existing language.

In Kay's *History of Programming* paper, "Early History of Smalltalk," he writes of being influenced by Marvin Minsky. Both Kay and Minsky were interested in the pedagogy of computer

---

[295] Ibid.

programming. Kay's primary focus when creating Smalltalk was to create a tool that a novice programmer could use to create their own applications. Through his focus on usability, Kay created a tool with wider ramifications. Smalltalk was seen by programming specialists as a programming environment that would simplify the broader problems of professional programming.

Confluence of Ideas

In the late 1960s, it was not only in the field of language design that the concept of object oriented design was appearing. Several areas of computing had practitioners approaching their problem set from an object oriented perspective. Computer aided design (CAD), hardware design, operating systems and networking were all using the object concept as an organizational method.

Sutherland's work on Sketchpad was influential both on programming theory specifically and the computer science community more broadly. Kay specifically referred to Sketchpad in his presentation at HOPL-II.[296] Sketchpad was an application, not a programming language. The purpose of the application is to produce images, using a light pen and tablet as ways of inputting and outputting information (communicating) with a computer. Sketchpad's

[296] Ibid., 514-515.

concepts are the foundations of CAD applications. Sketchpad uses a common sense concept of objects, but expands on that to store information with those objects: the mathematical formula on which they are based, the key axis points needed to recreate the object and links (or co-ordinates) of other objects in the displayed picture are saved with the object. Examining the graph representing the generic structures used to make an image in Sutherland's *ACM* article suggests these similarities with the eventual object oriented paradigm.[297]

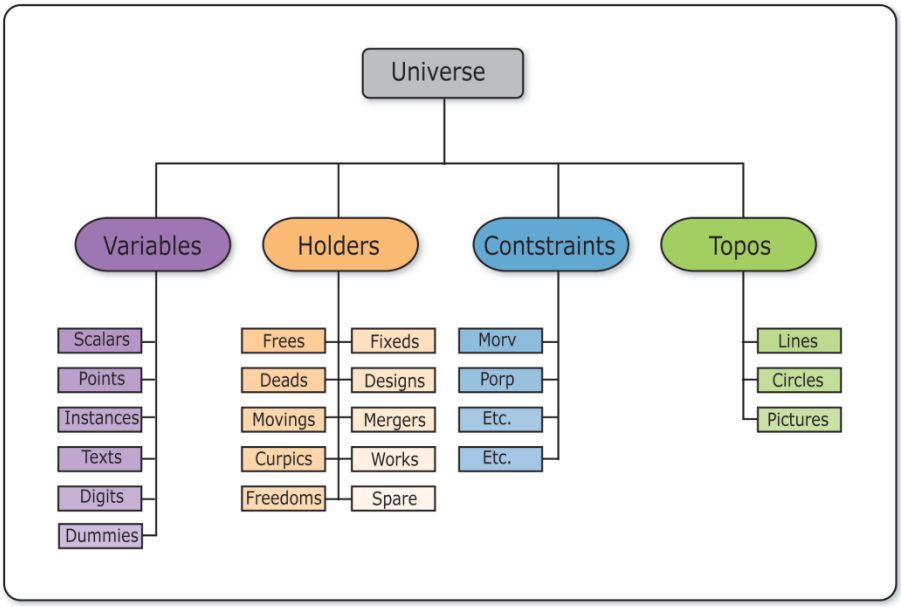


Figure 24: Sutherland's diagram of the Sketchpad structures

Sutherland's Sketchpad was not the only area of computing outside of programming language design to begin using an object

[297] Ivan E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *Proceedings of the AFIPS 1963 Spring Joint Computer Conference, May 21-23 1963,* (Baltimore: Spartan Books, 1963).

oriented framework. As early as 1961 the U.S. Air Force was thinking about files as modules, where the file was broken down into three sub-parts: the data in the file, the procedures for manipulating the files, and an array of pointers to those procedures. This modularization allowed the files to be transported from one location to another.

This idea, of using an object-centered world view in operating systems, was expanded by Hoare in "Monitors: An Operating System Structuring Concept."[298] Hoare used the term "monitor" to refer to the concept that we, universally, call a class. The main function of operating system design in the early 1970s was to create algorithms that would schedule the use of the computer's resources. Hoare wanted to create modules (monitors) that would work together to schedule each of the different types of resources. Each monitor would include the administrative data for the resource and the procedure and functions that programs would need to use when scheduling their use of resources. Hoare concluded in the article that the notation for these monitors could be based on Simula 67's "class" notation.[299]

By the early 1970s, computer architecture was also using concepts that resemble the object oriented methodology in the design

---

[298] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM* 17, no. 10 (1974).

[299] Ibid.

of computer architecture. The PDP-11 family of computers had one of the first orthogonal instruction sets. This is an instruction set where all the instructions have the same format. In orthogonal instruction sets the registers and addressing modes can be used interchangeably.

In a sense the registers and addressing modes can be seen as polymorphic – the registers accept the information regardless of its format and manipulates it appropriately. It is the abstraction of data and instructions that allows the registers and addressing modes to be used interchangeably and treat the data and instructions identically. Both of these concepts are fundamentals of object oriented programming.[300] Furthermore, the PDP-11 was intended to support time sharing. It had a two-level architecture, where, when running in supervisor mode, instruction addresses correspond directly to physical memory. In user mode, addresses are translated to physical memory. This user mode is a higher level of abstraction than earlier computer architectures.[301] Nor was DEC the only company incorporating elements of the object oriented world view in their architecture. By

---

[300] C.G. Bell, C. Mudge, J. McNamara, *Computer Engineering: A DEC View of Computer Design* (Bedford: Digital Press, 1978).

[301] C.G. Bell, *et al*., "A New Architecture for Mini-Computers – The DEC PDP-11," in *AFIPS Conference Proceedings 1970 Spring Joint Computer Conference, May 5-7, 1970* (New Jersey: AFIPS Press, 1970), 657-675.

1988, IBM had integrated the data hiding concept into its computer architecture and was describing their computer in terms of objects.[302]

The frequency with which object oriented ideas were appearing is further visible in a 1971 paper by Peter Denning, "3rd Generation Computing Systems," where he describes methods of modularization, information hiding and data abstraction as requirements of a 3rd generation system.[303] In some senses, the concept of ideas oriented around real world objects is as old as the analogue machines in use prior to the ENIAC. While the computing field matured, disciplinary boundaries were porous and ideas from different disciplines converged. There was little debate over the shift in worldview necessary for the use of object oriented programming, likely because these ideas were pervasive within the broader computing field.[304]

---

[302] The AS/400 was divided into layers: each level of system function was accessed through an opaque interface. This is analogous to the definition of data hiding in object oriented programming, where the object has a public interface, with private functionality. This use of data hiding prevented inadvertent modification of the underlying system specifications. For more information, see Arthur Norberg and Jeffrey Yost , "IBM Rochester, A Half Century of Innovation," *IBM 2007*, 36-37.

[303] Peter J. Denning, "Third Generation Computer Systems," *ACM Computer Systems* 3, no. 4 (1971).

[304] Further evidence of the influence of the object oriented worldview can be seen in the inception of object oriented databases. These object based data structuring schemas came to fruition in the late 1970s and early 1980s. Hurson, A.R.; Pakzad, S.H.; Cheng, J.-B., "Object-oriented Database Management Systems: Evolution and Performance Issues," *Computer* 26, no. 2 (1993): 48-58, 60. For more information see the Richard Bachman Archives at the Charles Babbage institute. Charles W. Bachman Papers (CBI 125), Charles Babbage Institute, University of Minnesota, Minneapolis, Minnesota

Parallel Development with Structured Programming

The most interesting element of the evolution of structured programming and object oriented programming is that not only do they develop in parallel; they originally appear as interchangeable - despite Dijkstra later stating in his private papers, "I don't think object-oriented programming is a structuring paradigm that meets my standards of elegance."[305]

Dahl was a co-author of Dijkstra's structured programming book. In that book, Dijkstra's chapter focuses on the elements specified in his early works — stepwise program composition and testing aids (mathematical induction, abstraction and enumeration) — to prove correctness. Dahl and Hoare's chapter on Hierarchical Program Structures can then be viewed in context as an example of how to structure a program. However, when seen in retrospect, the Dahl and Hoare chapter outlines the foundation of object oriented programming – the use of classes and objects as instances of classes.[306]

An example of Dahl's definition of class will demonstrate the similarity between our modern definition of classes and Dahl's 1970s

---

[305] Edsger W. Dijkstra, "Computing Science: Achievements and Challenges," EWD 1284, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

[306] O. J. Dahl , E. W. Dijkstra , C. A. R. Hoare, *Structured Programming* (New York: ACM Classic Books, 1972).

definition. Dahl defines concepts as a class of specialized instances. So, to use his later example, a histogram is a concept, and there are many instances of different histograms. This seems a general common sense idea that we may all agree on, but Dahl moved beyond to expand the definition of class. Dahl defined the term attributes as variables, procedures, or parameters declared as local to the class. He also defined that a call to a class generates a new object of that class. Using Simula 67, Dahl then describes a class generically:

(class declaration) = class (class identifier)

(formal parameter part) ; (specification part);

(class body)

(class body) = (statement)

Dahl used a specific example of a class that would create a histogram. When we compare Dahl's histogram class example to the earlier conception of a class, the structure is the same.[307]

[307] Ibid., 182.

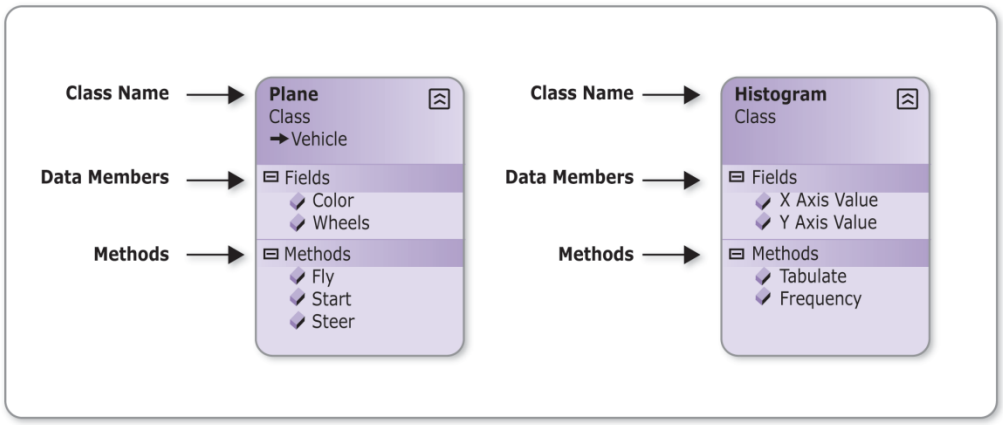


Figure 25: Comparison of the modern and historic class concept
Left Side: Modern Conception of Class, Right Side: Dahl's 1967 Conception of Class

Dahl and Nygaard also introduced the concept of Inheritance, although at that time they referred to this simply as "sub-classes."

The shift in perspective from organizing processes to re-imaging programs in terms of classes elevates Simula 67 from an implementation of structured programming to a new perspective on program design. Moreover, the detailed definition and real world implementation of these concepts are the foundation of the new programming methodology.

Evolution of the Fundamental Concepts of Object Oriented Programming

Like the earlier structured programming and automatic coding methodologies, object oriented programming evolved. Encapsulation, recursion, and polymorphism were not originally included as part of the definition of object oriented programming. In fact, they were well-entrenched concepts in programming long before Dahl, Nygaard, and

Kay defined the object oriented methodology. However, as the community of programming specialists refined the object oriented programming methodology these concepts became requirements of object oriented programming languages. This is similar to Mills' influential redefinition of structured programming, when he proposed a limited use of control structures as an additional feature of structured programming.

To explore the evolution of object oriented programming, I am going to briefly explore the history of each of these foundational concepts and examine how they were included in the definition of the object oriented programming methodology.

Encapsulation

Encapsulation is a loaded concept - it requires a programming environment that can support data hiding. Smalltalk was the first language to provide support for encapsulated objects, where the object has a public interface, with private functionality. By the time C++ was being developed all object oriented languages made use of encapsulation, but not all languages using encapsulation were object oriented. Ada provided encapsulation in its Package concept.[308] The

[308] William E. Carlson, et al., "Introducing Ada," in *ACM '80: Proceedings of the ACM 1980 Conference, January 1980* (New York: ACM, 1980).

structured, not object oriented, Modula-2 (the language Nicklaus Wirth created after Pascal) also provided encapsulation in its modules.[309] Oddly, when Bjarne Stroustrup, the creator of C++, one of the most popular object oriented programming languages, discusses encapsulation, he doesn't turn to the programming languages he used that featured forms of encapsulation. Instead Stroustrup turned to operating systems with which he had previously worked that used data hiding principles, like the PDP-11 that had the two-level architecture, with users and supervisors having access to different levels of data.[310]

Abstraction

In object oriented programming the term abstraction (the use of a concept or idea not associated with any specific instance) most often refers to abstract data structures – classes are an abstract data structure. This was also a well-established concept in programming by the time Stroustrup was working on C++. Pascal and ADA both use abstract data structures, mathematical models that describe a type of data with similar characteristics. One example is a stack[311] data type:

---

[309] Niklaus Wirth, "Modular-2 and Oberon," in *HOPL-III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, June 9-10, 2007,* (New York: ACM, 2007).

[310] Bjarne Stroustrup, *The Design and Evolution of C++* (New York: ACM, 1995).

[311] Stacks are an abstract data structure, but they have been around since the 1950s when they were first created by Frederich Bauer.

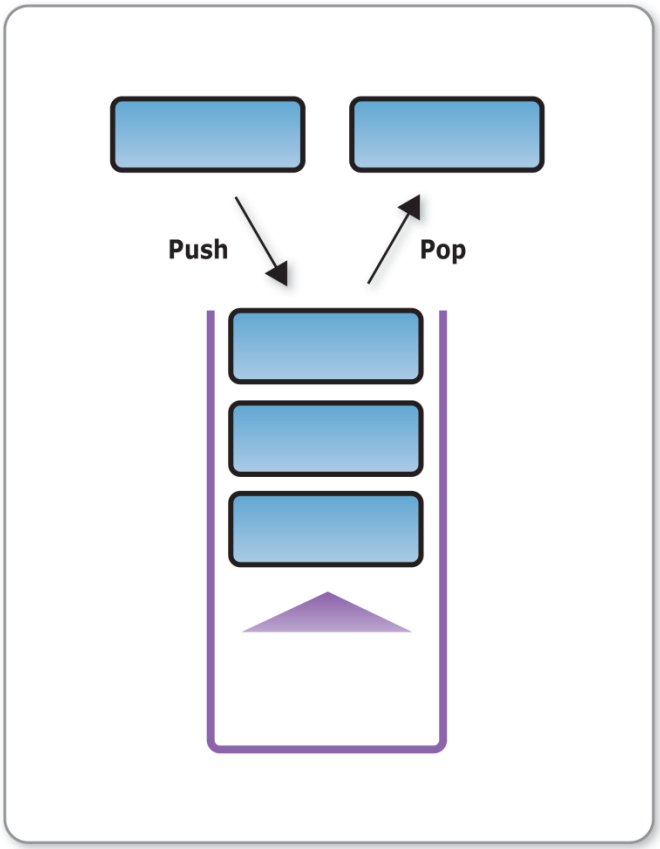


Figure 26: Graphical Description of a Stack

A stack is a last in, first out (LIFO) data structure. Inside the stack, the elements can be any type of data, which is why it is abstract, but it is governed by the push and pop mechanism of inserting and retrieving information. Data is pushed on to the top of the stack, and when a programmer retrieves data from the stack it is "popped" off the top of the stack, with the last entry being retrieved first. Evaluating an expression written in postfix notation (Polish notation) is commonly used as an example to show how a stack functions.

Creating a stack (even in languages that do not support the construct explicitly) can be accomplished by a programmer, using pointers to memory. However, it is much easier in high level languages that support abstract container classes. So, in a language where abstract data structures are supported by the environment, the declaration of a stack in C++ would look as simple as this:

stack<T> s;

where T is any kind of class – integers, strings, any type of data.

To create your own data structure, Stack, in C++ would be more like this:

```
class stack {
  int stck[SIZE];
  int tos;
public:
 stack()
 {
    tos=0;
 }
   void push(int i)
   {
    if(tos==SIZE) {
      cout << "Stack is full.\n";
      return;
    }
    stck[tos] = i;
    tos++;
   }
   int pop()
   {
    if(tos==0) {
      cout << "Stack underflow.\n";
      return 0;
    }
    tos--;
    return stck[tos];
   }
 operator int() {
   return tos;
 }
};
```

```
int main()
{
  stack stck;
  int i, j;
  for(i=0; i<20; i++)  stck.push(i);
  j = stck; // convert to integer
  cout << j << " items on stack.\n";
  cout << SIZE - stck << " spaces open.\n";
  return 0;
}
```

So, while it has become necessary for all object oriented programming languages to support abstract data structures, not all programming languages that support abstract data structures are object oriented. Pascal, which was not object oriented, clearly supports a number of abstract data structures — sets, arrays and records — but it doesn't support some of the more advanced stacks or link lists.[312] LISP (the list based language discussed in Chapter 2, which is not object oriented) supports arrays and abstract structures.[313] This illustrates that while LISP and Pascal support abstract data structures, neither of them are object oriented languages.

---

[312] Niklaus Wirth, "An Assessment of the Programming Language PASCAL," in *Proceedings of the International Conference on Reliable Software, April 21-23 1975* (New York: ACM, 1975).

[313] Steele, Jr. and Gabriel, "The Evolution of LISP."

Polymorphism

Polymorphism is another concept that has evolved to become integral to the object oriented methodology. Stroustrup argued in his article, "Why C++ is Not Just an Object Oriented Programming Language," that polymorphism was a required element of an object oriented language. He went on to demonstrate that this is not just true of C++, but of all languages defined as object oriented - Ada95, Beta, C++, CLOS, Eiffel, Simula, and Smalltalk. He also illustrated that it is not true of classical languages, using C and Pascal as his examples.

However, this argument depends on a sophisticated definition of polymorphism. Polymorphism is most generally considered the characteristic of being able to assign meaning based on the context, allowing an entity such as a variable, a function, or an object to have more than one form.

Stroustrup refined this definition. Stroustrup addresses only objects, requiring the language to provide the same interface to objects with differing implementations. A real world example of where a non-programmer may see this is in a print function. To a computer user there is one interface for print, but there may be an object "print" for each of the printers installed on the computer, each with functions

that communicate with the printer using the correct attributes of that printer.

But, prior to object oriented programming polymorphism was more commonly defined as functions or operators that can accept arguments of different types and behave differently depending on the type of argument they receive. This was often just called function or operator overloading. At the inception of automatic coding systems, beginning at least as early as FORTRAN, polymorphic traits were being included in the language design. Mathematical operators like + were "overloaded" where the operator could accept integers, real and complex numbers and treat them appropriately.[314]

While object oriented programming has used polymorphism with more sophistication, it was not created as a part of the object oriented methodology. Instead, the definition of polymorphism evolved with the object oriented methodology and became integral to it, further separating the object oriented programming methodology from the structured programming methodology.

Recursion

---

[314] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, R. Nutt, "The FORTRAN Automatic Coding System," *Proceedings of the Western Joint Computer Conference, February 26-28, 1957*, (New York: Institute for Radio Engineers, 1957).

Recursion as a concept began originally in mathematics. This concept was then refined and revised in computer programming, in parallel with the introduction of high level programming languages. Again, recursion is not a part of the concept of object oriented programming and was part of the theory of computation prior to the mechanization of computation, but because of its usefulness, it is required to be supported by all high-level programming languages, including object oriented languages, and is therefore subsumed into the methodology.[315]

The Adoption of Object Oriented Programming

By the late 1990s, object oriented programming became the most widely used worldview in use by programming specialists and vocational programmers alike. However, neither Smalltalk nor Simula 67 made a particular impact in the vocational programming literature. I argue that the programming language C++, designed in the early 1980s by Stroustrup while working at Bell Labs, was the impetus for the acceptance of the object oriented methodology.

What Is C++?

C++ is a compiled, general-purpose programming language, with particular support for the object oriented methodology. It

[315] R. I. Soare, "Computability and Recursion," *Bulletin of Symbolic Logic* 2 (1996): 284-321.

combines features of both high and low level languages. High level languages use abstract data structures and remove the programmer from needing to know the details of the functionality of the underlying machine. The syntax usually has natural language features, contributing to ease of use. Low level languages provide little abstraction from a computer's instruction set (or architecture), but low level languages also do not need a special compiler to run, and therefore run faster. Assembly code is a low level language. The reason C++ is said to combine these features is because C++ (and C) allow the programmer to directly access the register and memory locations of the computer, and manipulate those values, as opposed to true high level languages that only deal with abstractions of those values.[316]

C++ was by no means revolutionary. The concepts contained in C++ were all well understood by programming specialists prior to the introduction of C++. However, C++ combined these elements in an efficient and familiar manner, which allowed for broader use of the object oriented methodology by vocational programmers. C++ is still the dominant programming language used in most of our current operating systems, including Windows Vista/7, Macintosh's OS X, and

---

[316] Herbert Schildt, *C++ The Complete Reference Third Edition*, (Osborne McGraw-Hill, 1998).

modules of both Unix and Linux, including the KDE desktop. MySQL, the most popular open source database is written in C++. Many applications are written in C++, including the popular Adobe Systems software, like Photoshop and Illustrator. Some of the most popular games are also written in C++, including World of Warcraft.[317]

Origins of C++

Like Alan Kay, Stroustrup was heavily influenced by Simula 67. During his work on his PhD, Stroustrup coded a program to simulate system software running on a distributed system in the Simula 67 language. Stroustrup writes of the elegance of Simula 67's organizational concepts and explains specifically how well these organizational structures scaled as the complexity of his program increased. Yet, he found when running the program that the implementation did not scale. The run time performance of the implementation made it impossible to gather accurate data from the simulation.[318]

Just after Stroustrup finished his dissertation, moving from Cambridge to the Computing Science Research Center of Bell Laboratories in 1979, he began working on a networking problem that

---

[317] Bjarne Stroustrup, *C++ Applications*, (2011) (http://www2.research.att.com/~bs/applications.html)

[318] Stroustrup, *The Design and Evolution of C++.*

required similar simulation capabilities – a tool that could simulate the modular structure of the UNIX system he was working with and the communication pattern between these modules. To complete this work, Stroustrup created a preprocessor for C that he called Cpre. This preprocessor added Simula-like classes to C and was eventually expanded to become C with Classes.[319]

C with Classes became very popular. By 1982 C with Classes had enough users that it had paid for itself, but it had become a burden to support. It did not have enough users to pay for that support and continued development necessary for a widely used programming language. Stroustrup (and Bell Labs) chose to develop a new language that would better serve a large enough set of users to pay for support and continued development.[320]

That language was C++. C++ was intended to provide language support for an object based organization of programs, like Simula 67, in conjunction with the efficiency, portability and flexibility for systems programming that had popularized C. While none of the language features alone were revolutionary, by providing a high level object oriented framework combined with the ability to produce low level

---

[319] Ibid.

[320] Ibid.

manipulation with the system and with an efficient run time the C++ language achieved significant popularity.

C++ as the Vehicle for Object Oriented Programming

C++, unlike Smalltalk and Simula67, brought object oriented programming into both the broader vocational programming community and the community of programming specialists. This can be measured in part by the following graph, demonstrating the significant number of articles published on C++ in the years after its introduction, compared to the articles written on Smalltalk in the years just after its introduction. Data was normalized to take into account the different number of total articles published in the differing time periods.

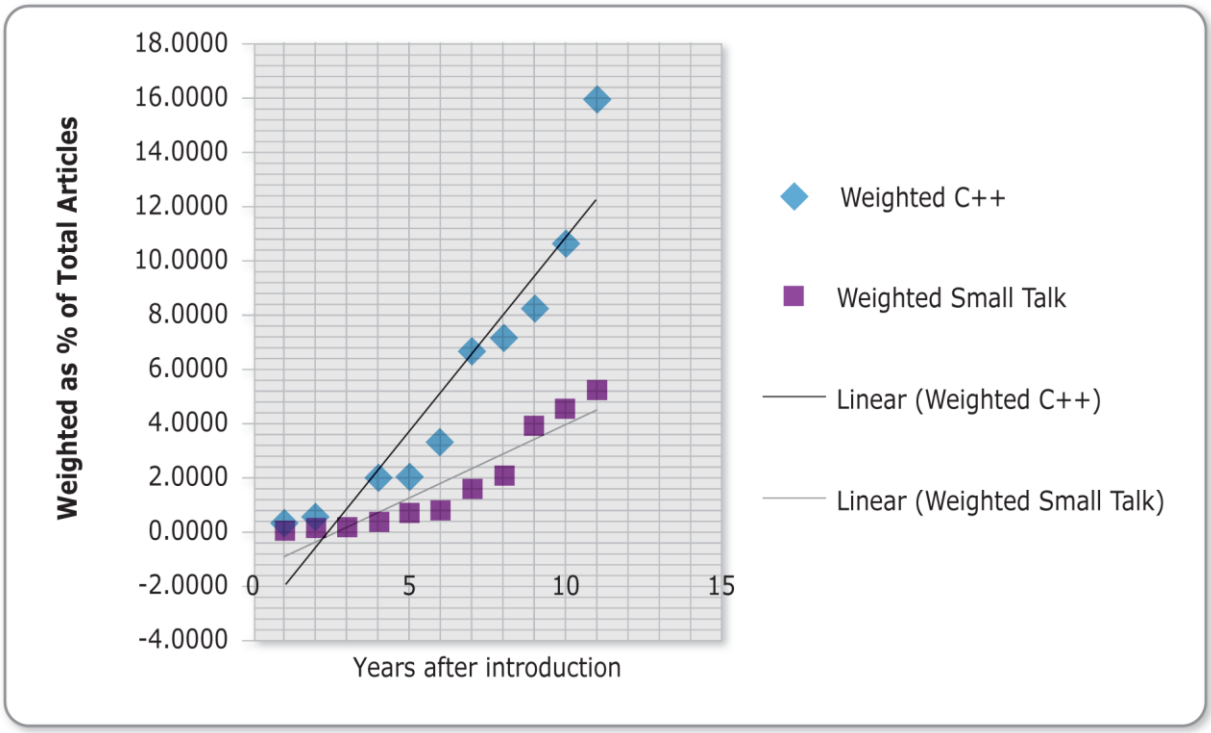


Figure 27: Results of Literature Review

The number of articles about C++ compared to the number of articles about Smalltalk.

The results were weighted to adjust for the difference of articles per year.

The popularity of C++ is also illustrated by a graph created by langpop.com, recreated as Figure 27. This graph demonstrates the popularity of C++ compared with a number of other languages. The graph illustrates the popularity of C++. langpop.com used a variety of search techniques to create a graph of the most popular programming languages on the World Wide Web. These search results were then normalized by DedaSys.[321] While these results are skewed, because

[321] According to lang.pop this combines and normalizes data obtained from several sources: 1. a yahoo api that searches for x programming language and then totals the mentions of each

they reflect popularity on the internet, when taken in conjunction with figure 26 and the literature, it is clear that C++ is a very popular language in the vocational programming community.

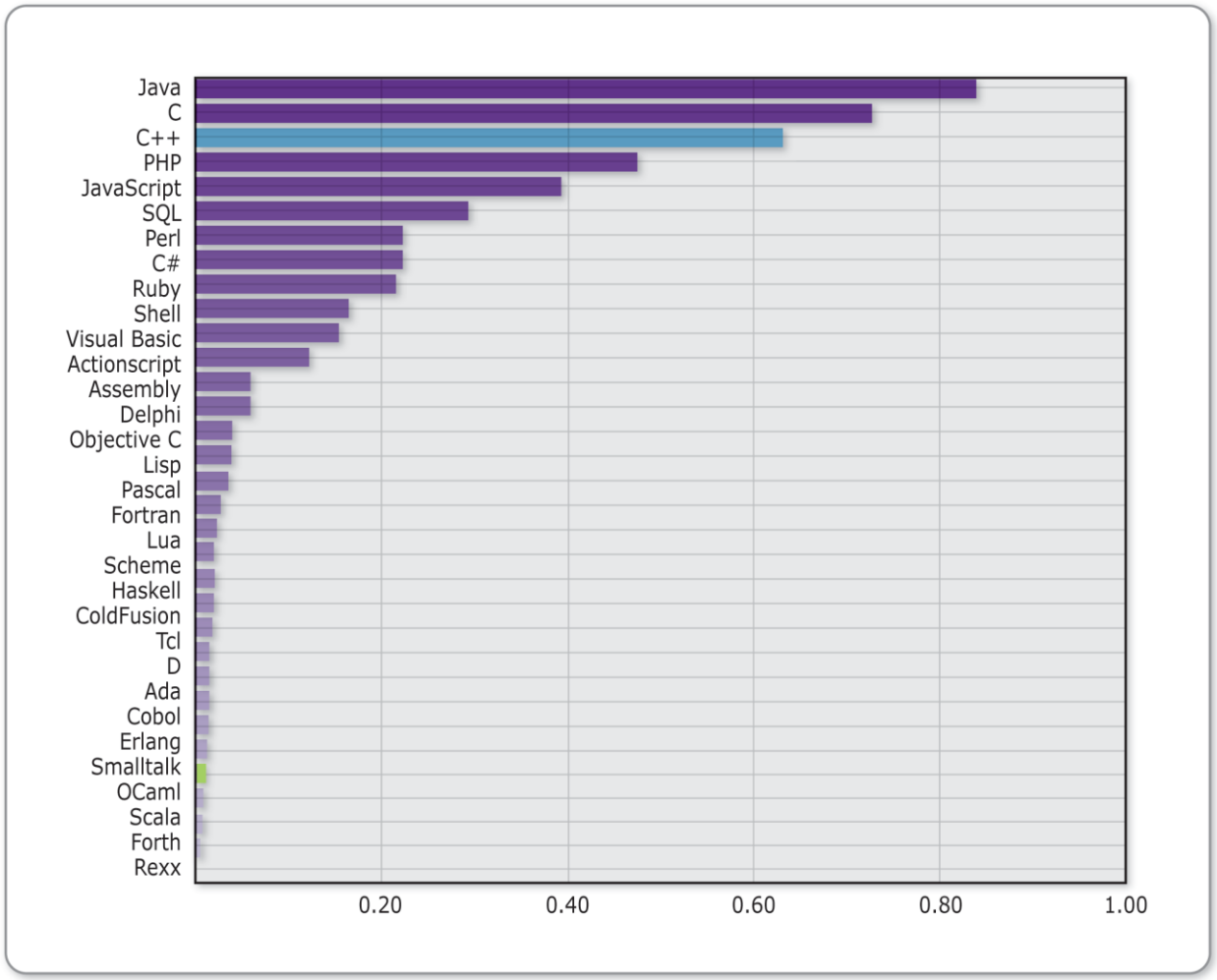


Figure 28: Popularity of C++, using data from langpop.com.
This graph is from materials collected in 2009.
Langpop.com is constantly updating and currently has 2010 data on their site.

---

instance of x, 2. Google's CodeSearch that finds public source code, and 3. The number of books being carried at Powell's books on each language.

Why was C++ so popular? We've already seen that it was not revolutionary. Object oriented programming was already being accepted as a useful theory. None of the features of object oriented programming were new concepts. Other languages, including the DOD-sponsored Ada, the expanded versions of Pascal and the redefinitions of ALGOL, were being developed that included these concepts.

In part, the commercial popularity of C++ was due to the existing popularity of C. C was written by Dennis Ritchie, but C was essentially an extension of Ken Thompson's B. Thompson was writing an assembler for a PDP-7. By 1973 the C language was designed and a compiler for the PDP-11 was written and functional. Bell chose to rewrite the Unix Kernel for the PDP-11 in C that summer. The changes following the 1973 definition were primarily focused on considerations of portability. As the UNIX operating system spread, the C language spread with it. Compilers for other computers, like the DEC VAX machines, were written quickly after the Unix Kernel was written. Ritchie states the C language was enormously popular. He argues that this is because the language is reasonably simple, but what really drove its popularity was its portability and the popularity of UNIX.

Anywhere UNIX was available, C was available and UNIX was (and is) an immensely popular operating system.[322]

C++ is based on C and is largely compatible with C. With a similar syntax and supporting most of C's features, it was able to be perceived as a "better" C. This allowed existing C programmers to learn C++ quickly, even using it as a superset of C without immediately grasping the object oriented concepts that differentiate C from C++. Because of this compatibility with a widespread, existing language, C++ was immediately commercially viable. Programmers did not need to undergo lengthy training sessions or sit idle while they learned a "new" language. Furthermore, vocational programmers are usually constrained by an existing system — hardware, operating languages, documentation styles — and C++ was now compatible with a broad number of those systems because of its compatibility with C.

Finally, C++ gained popularity because it simplified the management of large scale programming projects. Strict encapsulation, enforced in the object oriented methodology, allowed for vocational programmers to be assigned small tasks, which could then be verified as correct and inserted easily into the larger software

---

[322] Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. eds., *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993,* (New York: ACM, 1996), 671-681; 685.

system. These small, encapsulated tasks could then be easily re-used in other software systems. Managers are then able to realistically assume that if the overall design of the system was secure, each component would work within the system to produce the desired results.[323]

Relationship of the Object Oriented Methodology to Complexity & Verification

Problems of complexity and verification continued to plague programming specialists after the adoption of structured programming. A plethora of languages were created to help impose a structured design on programmers. Pascal and Ada are thought of as structured languages and were written in response to the discussion of complexity and verification that was threaded through the structured programming debate.[324]

Pascal was developed by Nicklaus Wirth concurrent with the structured programming debate. When Wirth speaks about Pascal he argues that he wrote it as a language that would be easy to write and able to be formally verified, and in many ways Pascal can be said to

---

[323] Doug Johnson, et. al. "Managing Object Oriented Projects," *Proceedings of the Tenth Annual Conference on Object Oriented Programming Systems, Languages, and Applications, October 15-19, 1995,* (New York: ACM, 1995).

[324] Niklaus Wirth, "Recollections about the Development of Pascal," in *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993,* eds. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., (New York: ACM, 1994), 340.

meet those goals.[325] However, the main critique of Pascal has been that, while it is a great language for pedagogical purposes, it is not a serious programming language. This criticism stems from many aspects of the language design. Many claim that this is no longer true, because of the extensions made to Pascal by both Borland and Apple. However, Wirth himself has criticized those extensions as moving Pascal away from its goals of usability and verifiability.[326]

Ada was also produced in a response to the problems of complexity and verification, but not by the computing community. Like COBOL, Ada was sponsored by the Department of Defense. In 1975, the DoD realized that they were spending three billion dollars a year on software written in 400 different languages and dialects. Ada's primary goals were to improve the reliability, readability and maintainability of programs. Ada's designer, Jean Ichbiah, felt that they had achieved that goal, arguing that Ada was read linearly and therefore had improved readability and decreased the complexity of software written in Ada. Ichbiah argued that this corresponded with Dijkstra's work on verifiability, because Ada reduced the dynamic dimension of the program's execution. However, Dijkstra criticized

---

[325] Niklaus Wirth, "An Assessment of the Programming Language PASCAL," in *Proceedings of the International Conference on Reliable Software, April 21-23 1975*, (New York: ACM, 1975), 323-330.

[326] Wirth, "Recollections about the Development of Pascal," 339.

Ada, arguing that it would not reduce software costs through standardization and it would not produce reliable code, because it was so complicated. Moreover, while Ada's supporters argue that its limited use was related to the expense of compilers and a lack of tools, in 1984 Ada was assessed in the "Ada Letters" as unreliable – this unreliability is far more likely the cause of Ada's limited use.[327]

I discuss these examples because they illustrate that, while language designers enforced structured programming techniques in their languages, the problems of complexity and verification were still an important part of the discourse on programming theory. During the mid-seventies, workshops on the quality of software were being held, workshops like the ACM's 1978 *Software Quality Assurance Workshop on Functional and Performance Issues*. Fred Brooks' *Mythical Man-Month*, which talks about the problem of adding labor to a problem as a solution, was published in 1975.[328] Constant discussion of the Software Crisis abounded. Problems of complexity and verification were still in the forefront of the literature in the late 1970s.[329]

---

[327] Albert Llamosi, Pere Botella, Fernando Orejas, "On Unlimited Types and Reliability of Ada Programs," *SIGAda Ada Letters* IV, no. 1 (1984).

[328] Brooks, Jr., *The Mythical Man-Month.*

[329] Between 1970 and 1975, the ACM has over 1,545 articles dealing with the problem of complexity in software. During the same time period there were 875 articles written on verification. The total number of articles published by the ACM between 1970 and 1975: 24, 515.

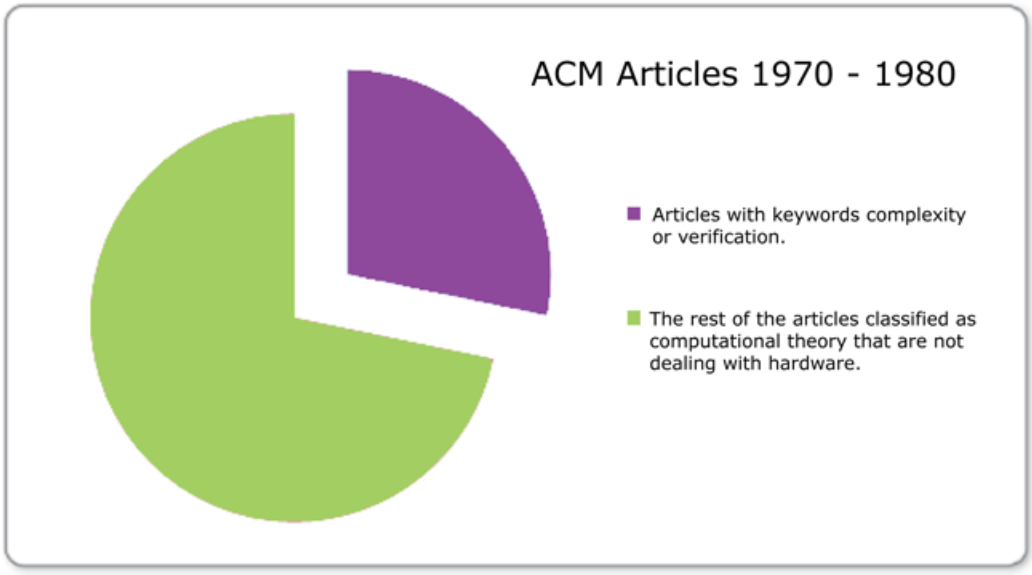


Figure 29: ACM Articles Related to Complexity and Verification
This is in relationship to the literature classified as computational theory that is not dealing with hardware
The scope of the review was limited to the years between 1970 and 1980

The object oriented methodology was created in response to problems of complexity. When interviewed by Barbara Ryder for a 2005 *ACM* article, Professor Boris Magnusson, who worked with Dahl and Nygaard on Simula 67 argued that his work while developing a large, complex, software system influenced his work on Simula 67.[330] Alan Kay talks often of a bet where he argued he could create "the most powerful programming language in the world"[331] in a single page of code.[332] The outcome of this bet was the creation of Smalltalk. The

[330] Barbara G. Ryder, Mary Lou Soffa, Margaret Burnett, "The Impact of Software Engineering Research on Modern Programming Languages," *Transactions on Software Engineering and Methodology (TOSEM)* 14, no. 4.

[331] Kay, "The Early History of Smalltalk," 532.

[332] Ibid.

inference here is that other languages may have also been powerful, but the complexity of their long definitions was detrimental.

At the first OOPSLA (Object Oriented Programming Systems Languages and Applications) Conference in 1986, David Moon described object oriented programming as a technique for organizing large programs. Moon argued that the object oriented technique makes it practical to deal with "programs that would otherwise be impossibly complex."[333]

Stroustrup's early articles also demonstrate that he was hoping to contribute to the solution of the problems of complexity and verification. In 1978, while still working on his PhD in Cambridge, Stroustrup published an article that discussed how a programmer could create a unified interface for different kinds of modules: processes, procedures and data modules. This interface would then hide the functionality of the module from the module's user. At the time, Stroustrup talked about the benefits of this for system programming, specifically parallel and distributed computing. In retrospect it is clear that this is the type of encapsulation that Kay was talking about and the type of encapsulation that Stroustrup

---

[333] David A. Moon, "Object Oriented Programming with Flavors," in *Proceedings of the 1986 Conference on Object Oriented Programming Systems, Languages, and Applications, 1986*, (New York: ACM, 1986), 1-8.

implemented in C++. Stroustrup explicitly argues that this type of encapsulation would ease programming and allow for code re-use. He argues that by allowing the modules to be abstract, and then to be called with variables to create the specific instance of the module, the module is then able to be reused.[334]

Object oriented programming has not been the silver bullet for problems of verification and complexity. Most leading proponents of object oriented programming concede that while it is a useful construct, there is still more work to be done. Hoare, who argued that object oriented programming was the natural scientific progression of the programming discipline, stated in the same interview that modern programming languages, like C++, are more complicated than they used to be because the object orientation, inheritance and other features are not thought through from a theory of correctness or from a scientifically based discipline.[335] Wirth remarked in a paper on pedagogy in 2002 that the software industry suffers from unwarranted complexity and a lack of regularity and reliability in programs.[336]

---

[334] Bjarne Stroustrup, "On Unifying Module Interfaces," *SIGOPS Operating Systems Review* 12, no. 1.

[335] Charles Antony Richard Hoare, Oral history interview by Philip L. Frana, July 17, 2002, OH 357. Cambridge, England, U.K. Charles Babbage Institute, University of Minnesota, Minneapolis.

[336] Niklaus Wirth, "Computing Science Education: The Road Not Taken," in *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, June 24-28, 2002,* (New York: ACM, 2002).

Stroustrup commented in an interview from the year 2000 that even after Brooks' *Mythical Man-Month* and an awareness of the problems from SAGE, when projects are in trouble, the commercial software community still simply throws more people at the problem. To improve the field, he argues that the commercial software community needs to take a more systematic approach to system development, deployment, and maintenance. Stroustrup argued that as a community, programming specialists needed to place a higher emphasis on correctness, quality, and security. Moreover, he argues that object oriented programming methodologies have not been elegantly combined with functional techniques.[337]

In this same interview, Gosling, the creator of Java (the language that some say replaced C++ as the most popular language) argued that his biggest current concern in the discipline is that the complexity of software systems is increasing and that the community of programming specialists does not know how to build big, complex systems. In 2000, he argued that one of the driving forces for the future of software was still complexity. The environments we expect software to run on have changed dramatically (in cell phones, smart

[337] Herb Sutter, "The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling," *Java Report* 5, no. 7 (2000); and the *C++ Report* 12, no. 7 (2000).

products, embedded systems, real time systems) and the way the programmers write programs must change with the environment.[338]

Impact of Object Oriented Programming on Complexity and Verification

Since before the electronic mechanization of computation, automated verification of algorithms has been a research topic. Prior to the software crisis and the corresponding increase of interest in verification of programs the discussion of formal, automated verification was generally limited to the artificial intelligence community and mathematicians.[339]

In the 1960s there was growing concern about the accuracy of software as a component of what was being called the software crisis. As we saw in the previous chapter, as hardware became less restrictive, there was a decrease in impetus to write clean programs, the focus was on simply finishing the project. As a result of the introduction of real time systems, time share applications and telecommunications, software applications were becoming more complex. The number of programmers in the field had increased significantly, complicated the communication and creating more

---

[338] Sutter, "The C Family of Languages."

[339] D. Mackenzie, "The Automation of Proof: A Historical and Sociological Exploration," *IEEE Annals of the History of Computing* 17, no. 3 (1995): 7-29.

structured organizational and managerial requirements. [340] With this increasing complexity, and software programs being applied to more crucial areas of application, there were increasing concerns about program accuracy.

In response to this, the idea of applying mathematic proofs of correctness to programs became a significant topic of research, first in the artificial intelligence community, but later by programming specialists. By the early 1960s through the work of McCarthy, Naur, Floyd and Hoare (all figures that featured prominently in the discourses on structured programming and object oriented programming) verification was a popular topic in computer science. As research on verification matured, there were several aspects of inquiry: specific to this dissertation – a schism between programming specialists who focused on applying mathematical proofs to algorithms to prove their correctness, and those specialists focused on pragmatic approaches that tested for program correctness.[341]

As early as the 1960s, it had become axiomatic in the community of programming specialists that testing alone could not

---

[340] C.B. Jones, "The Early Search for Tractable Ways of Reasoning About Programs," *IEEE Annals of the History of Computing* 25, no. 2 (2003).

[341] Mackenzie, "The Automation of Proof," 7-29.

ensure program correctness.[342] By the late 1980s it had also become dogma that mathematical verification of code was not applicable to complex, real world software implementations.[343] As a result, program verification evolved to become the use of design methodologies, like stepwise design and object-oriented design. In these more formal design methodologies, assertions were used to justify design choices during the construction of the design, by modularizing the program components. [344] The use of assertions to provide exception handling, a construct designed to handle the occurrence of special conditions that change the normal flow of program execution and error checking has almost replaced formal verification techniques that "prove" an algorithm is correct.[345]

By the early 2000s, when the object oriented programming methodology was at its peak, we can see that the work being conducted on verification is very different to the work being done in the 1970s. The more recent discussion was about logically verifying the model of the software to be sure that design performs the function

---

[342] Jones, "The Early Search."

[343] James Fetzer, "Program Verification: The Very Idea," *Communications of the ACM* 31, no. 9 (1988)

[344] Jones, "The Early Search."

[345] Fetzer, "Program Verification: The Very Idea."

[345] Ibid.; Mackenzie, "The Automation of Proof," 7-29.

for which it was intended.[346] Even the DoD's infamous "Orange Book" didn't suggest mathematical verification techniques.[347]

With the popularity of object oriented programming, verification has essentially become a series of techniques. With a non-sequential program execution, mathematical verification of the entire program became virtually impossible. Instead, programmers rely on verifying the program's design, refined testing techniques that focus the testing on components (not the entire system) and the use of assertions to provide exception and error handling in case of failure.

I argue that this combination of techniques no longer focuses on the problem of verification directly. Instead, the community of programming specialists has focused on reducing the complexity of designing a software system and reducing the complexity of testing that system by breaking it down into its component parts. This has further benefits in a commercial programming setting, because it allows the division of labor to be instituted both in the creation of the software and in the testing and maintenance of the completed project. This division of labor has inherent benefits for the management of a large scale software project. What has remained from the verification

---

[346] K. Rustan, M. Leino, Greg Nelson, "Data Abstraction and Information Hiding," *Transactions on Programming Languages and Systems (TOPLAS)* 24, no. 5 (2001).

[347] Jones, "The Early Search," 19.

discourse of the 1970s isn't the mathematical verification of programs, but the use of mathematical assertions to conduct error checking and exception handling – in effect providing failsafes for when the verification of design and testing techniques fail.

Relationship of OOP to Other Programming Paradigms

The enigma of object oriented programming is that when seen in the large it is a shift from viewing programming as describing processes to viewing programming as communication between objects, it is a radical innovation in both the way programmers understand programming and in the methodology of designing a program. However, when you examine object oriented programming as a collection of its component parts, it seems less revolutionary and instead can be seen as a straightforward use of good programming practice.

A number of luminaries in programming theory view object oriented programming in this way. Donald Knuth, author of *The Art of Computer Programming*, has insisted that he has always thought of programming using the techniques that have been subsumed into the object oriented paradigm, but the languages he has used don't enforce that discipline. Instead, the discipline comes from Knuth as a

responsible programmer. He argues that the change is that the new programming environments enforce the rules he would normally use:

> *I've always thought of programming in that way, but I haven't used languages that help enforce the discipline; I've always enforced the discipline myself in other languages.*[348]

Nicholas Wirth has a similar view, although he expanded on it in more detail. Wirth argued that programming methodologies had become a religion in computer science. Devotees of structured programming deny the import of object oriented programming; while object oriented programmers evangelize about its merits. In an interview, Bjarne Stroustrup expounded a similar view. In that interview he argued that there are programming approaches that work best in particular domains, but that this idea, that there isn't one best programming approach is difficult for the community of programming specialists to accept.[349] This concept is seen throughout the history of computing, with the universal language concept that drove COBOL and ALGOL and the continued comparison of programming methodologies to Kuhn's revolutionary paradigm.

Wirth argues that this divide between the methodologies is a commercial fallacy. I think viewing this divide only as a method for

---

[348] Dan Doernberg, Interview with Donald Knuth for the Computer Literacy Bookshop, December 1993, http://tex.loria.fr/litte/knuth-interview

[349] Will Tracz, "The Real Stroustrup Interview," *Computer* 31, no. 6 (1998): 110-114.

selling the programming environments downplays the rancor that is visible in the literature. Instead, I see it as a combination of elements. Selling new programming environments, like Turbo Pascal and Borland C++ provides part of the explanation for the emphasis on new methodologies as revolutionary improvements. But I also see a divide in the literature between the young Turks and the established scientists.

Wirth and Knuth are tied to their own advances in computing, advances that are heavily related to the mathematization of programming, the use of algorithms and the advances of structured programming techniques. Wirth bluntly argues that object oriented programming has not added new concepts. In his mind it just emphasizes the binding of procedures to objects and the use of inheritance. As we have seen, there is some truth to this – encapsulation, abstraction, recursion, and polymorphism were not new phenomena. Yet, object oriented programming changed the way programmers perceive programs, changing the programming worldview, as well as bringing the fundamental concepts to a significantly broader audience.

I think what Wirth is missing about object oriented programming is laid bare in an interview answer: "Static

modularization is the first step towards OOP."[350] What this answer reveals is that modularization does not require programmers to view their programs as objects communicating; instead for Wirth, the program is still seen as a representation of processes. The first step towards object oriented programming is the shift in world view from the primacy of objects, over processes –a shift that Wirth had not made.

However, Hoare (who is from the same generation of programmers as Wirth and Knuth) does take this broader view of object oriented programming, seeing it as a shift from processes to objects. Hoare sees the change in programming methodologies as paralleling his own scientific progress moving from sequential code with hierarchical construction towards an object oriented worldview and concurrency – the simultaneous execution of processes that potentially interact.[351] The difference here is not one of chronological generations - it is because Hoare was intimately involved in the birth of object oriented programming, co-writing the chapter with Dahl in 1972.

---

[350] Dr. Carlo Pescio, "A Few Words with Niklaus Wirth," *Software Development* 5, no. 6 (1997).

[351] Charles Antony Richard Hoare, Oral history interview by Philip L. Frana, July 17, 2002, OH 357. Cambridge, England, U.K. Charles Babbage Institute, University of Minnesota, Minneapolis.

In 2002, Bertrand Meyer, who created the object oriented Eiffel language also saw object oriented programming as the future of computing, calling it the "only way that made any sense." When looking at the field, Meyer argues that "the object-oriented mode of thinking has won. When it comes to serious, professional development, there is no credible alternative".[352] This differs so significantly from the positions of Dijkstra, Knuth and Wirth that it illustrates the divide in how programming specialists view the impact of object oriented programming.

Stroustrup also speaks positively about object oriented design, arguing that it tends to lead to better code from a procedural (process centered) approach. He argues that studies[353] have shown code written in the object oriented methodology creates more flexible, extensible and maintainable code.

Object Oriented Programming Today

Object oriented programming is the dominant program design methodology of the software community, both in programming theory

---

[352] Bertrand Meyer, "Interview with Programming Expert Bertrand Meyer", November 30, 2001, *InformIT* (provided courtesy of Prentice Hall PTR), http://se.ethz.ch/~meyer/publications/interviews/informit_interview.html, Accessed July 27, 2010

[353] Note that the "studies" are left very vague – Stroustrup suggests that AT&T has conducted studies that support this, but since they have not provided access to their archives, there is little room to prove this statement.

sub-set but also in the commercial programming field. Object oriented programming was developed in the 1960s, in response to the problems of complexity and verification that we saw detailed in Chapter 4. Despite developing in parallel with structured programming, object oriented programming was not well accepted until the 1980s when the programming language C++ provided a vehicle for popularization. As the object oriented world view was slowly adopted, structured programming became less relevant to cutting edge software programming.

However, structured techniques are still in use, as are the ad hoc techniques that existed before the introduction of structured programming because of the many legacy systems that are still in existence. Legacy systems are those systems that, while created with obsolete methods, languages or hardware, still function to fulfill the needs of their users. Legacy systems are not uncommon. These systems need to be either maintained and updated, or, at times, replaced to meet customer needs.

In some instances these legacy systems are completely replaced. For example, in 1999 New York State began the tedious task of replacing their 17-year-old legacy accounting system. The system was still stable, but it could no longer fulfill the needs of their

users in terms of access, flexibility and automation.[354] Currently, the new system (begun in 1999) has still not gone live - 11 years later.[355]

However, in other instances, the legacy system is simply black-boxed from all but the IT personnel and encapsulated with newer technology. For example Wells Fargo maintained customer accounts on a number of different systems, depending on the type of the account. These systems range from IBM mainframes to VAX systems. To find all of a customer's accounts the employees would need to access each different system using terminal emulators without a graphical user interface. In 1995, Wells Fargo introduced middleware, which would accept input from the employees, and then request information from the legacy systems. This eases the burden on the employees, but even now, Wells Fargo still relies on legacy systems.[356]

Working on, or replacing, legacy systems require programmers to be intimately familiar with their (long obsolete) languages, methodologies and hardware systems. Until all of these systems are replaced, the techniques of structured programming and the syntax of languages from the 1960s (and earlier) will not be forgotten.

---

[354] I Fisher and M. Bradford, "New York State Agencies: A Case Study for Analyzing the Process of Legacy System Migration: Part I," *Journal of Information Systems* 19, no. 2 (2005): 173-189.

[355] "Office of the New York State Comptroller," http://osc.state.ny.us/agencies/cas/index.htm

[356] Erik S. Townsend, "Wells Fargo's 'Object Express'," *Distributed Object Computing* 1, no. 1 (1997): 18-27.

While object oriented programming was a radical shift in the world view of programmers and is currently the dominant design methodology, programmers will have legacy systems based on earlier methodologies and programming languages for some time to come. Moreover, object oriented programming is not the end of the story. The ballooning popularity of the internet and the World Wide Web resulted in a need for new dynamic languages. These languages need to be platform independent and customized for use over the internet. These languages are incorporating the object oriented methodology and extending its fundamental concepts to new domains of interest.

Object oriented programming was not a silver bullet that solved the problems of complexity and verification. Since the 1960s, it has become unfashionable to assume a universal solution to problem solving in software. The entire definition of verification has been reformulated to include a variety of techniques, both in the design stage and during the testing phase. Increasing complexity is seen as the cost of faster, larger software projects and the battle to decrease complexity in programming continues to rage. However, the software field no longer hopes for a silver bullet and instead, focuses on the best practice for each project type.

Chapter 6: Analysis and Conclusion

Analysis

This dissertation addresses changes in programming theory between the 1950s and the mid-1980s. After the mid-1980s, plurality has become the dominant theme in programming theory - plurality in worldviews, plurality in methods, and plurality in languages. This emerging doctrine of plurality superseded the field's previous goal of finding a single universal solution. Moreover, the emergence of this doctrine resolves several tensions that long pervaded the field.

Complexity and Verification in 2010

This new landscape, with its embrace of multiple worldviews, design methods, languages and systems, has dramatically increased the complexity of the computing field. Many of the new methodologies, design methods and languages are claiming to decrease complexity, but the sheer number of paths to choose from when approaching a problem must be daunting for programmers.

Moreover, in this new landscape, it seems that the need for flexible programs that work across different computing platforms has become its own stimulus for change in the methods of programming software. The software projects being attempted increase the

complexity programmers are faced with. Many of these new programs need to be able to communicate across numerous systems with different operating systems and applications, all of which were created by different people and organizations. Clearly, the adoption of object oriented programming did not solve the problem of complexity.

Verification is also still at the forefront of the field. However, it too has been redefined to embrace the multiplicity of methods. Often, verification is now described as quality assurance, and as we saw in Chapter 5, even in the early 1980s verification had already been redefined, moving past the debate that pitted mathematical verification techniques against practical testing procedures. Mathematical verification of software is virtually unheard of in 2010. Indeed, even the term verification is rarely used, with the term "quality assurance," a phrase originally found in manufacturing, gaining increased popularity. Verification, or quality assurance, now focuses on using assertions and exception handling, combining mathematical (mostly statistical) analysis and a variety of testing techniques throughout the development cycle of the program.

Resolving Tension

One tension, seen throughout this dissertation, which is resolved by this doctrine of plurality is the tension between programming worldviews and programming methods. This dissertation clarifies that

programming worldviews (or, as often called in the field, paradigms) are fundamental styles of programming. Object oriented programming is a shift in worldview, a change in the fundamental style of programming. All programming prior to the adoption of object oriented programming was conducted with a procedural (or, as it is sometimes called, the imperative) worldview. Alternatively, programming methods are the tools used to design and create programs. Stepwise or top down design (seen in chapter 4) are examples of programming methods.

The tension between these two concepts stems, in part, from the hyperbole in the literature of the field. Structured programming was regularly described by its advocates in terms that suggest it is a shift in the programming paradigm or worldview. However, in chapter 4, I illustrated that this was not the case. Structured programming was a set of "best practices" for the existing procedural paradigm. Alternatively, I have shown that object oriented programming was an actual shift in the worldview of programmers.

State of the Field in 2010

Plurality of Paradigms

In 2010 there are many programming worldviews. While the object oriented worldview is still dominant, the process-driven programming style (the worldview that underlies both the ad hoc

methods of the earliest programs and structured programming) is still in use.[357] The newer concurrent programming worldview (supporting multi-core processors) and the declarative worldview (programming that expresses the logic of a computation without describing its control flow) are among a number of new programming worldviews, all of which are now used by programmers to solve different types of programming problems.

Concurrent programming is an interesting example of this new multiplicity of computing styles, because it also illustrates the changes in hardware. Since the mid-1980s, in the hardware field there has been an increasing popularity of multi-core processors.[358] Programs for multi-core processors need to be optimized to use these additional resources. As a result, concurrent computing has flourished, spawning many design methods and environments that support concurrent programming. Petri Nets is one example of the new environments that support concurrent programming. It is a mathematically oriented, graphical environment that is used to specify and analyze the computer system, aiding program design.[359] After the program is

---

[357] Although now it is called the procedural or imperative programming worldview.

[358] The core is the part of processor that reads and executes instructions. Originally applied to mainframes, these multi-core processors are now being applied to personal computers.

[359] As an aside, this use of graphical environments used to aid program design has also become very popular in all of the programming paradigms.

designed, the code is implemented in programming languages that are optimized to support concurrent programming.[360] Concurrent Pascal and Modula-3 are two well-known examples of languages that support concurrent programming.[361]

Declarative programming argues that a program can be viewed as a theory of formal logic. Computation is then viewed as deductions in that logic system.[362] Declarative programming is rooted in the 1960s work on Artificial Intelligence, specifically the theorem proving work with which John McCarthy was involved (described in Chapter 4). A future topic for research may be to explore the evolution of declarative programming, against the backdrop of the structured programming controversy and the emergence of object oriented programming.

The difference between declarative programming and other worldviews is that in declarative programming, the programmer states (declares) what is to be achieved and leaves it up to the system to get the job done. An example is the use of HTML. HTML only describes

---

[360] Glynn Winskel and Mogens Nielsen, "Models for Concurrency," in *The Handbook of Logic in Computer Science Vol. 4: Semantic Modeling,* eds. S. Abramsky, Dov M. Gabbay , and T. S. E. Maibaum (Oxford: Oxford University Press, 1995), 1 - 148

[361] Peter Denning, "Computer Science: The Discipline," in *Encyclopedia of Computer Science*, eds. A. Ralston and D. Hemmendinger (Chichester: John Wiley and Sons Ltd., 2003), 405-419.

[362]Olof Torgersson, "A Note on Declarative Programming Paradigms and the Future of Definitional Programming," March 19, 1996, http://www.cse.chalmers.se/~oloft/Papers/wm96/wm96.html, Accessed November 17, 2010

what should appear on a webpage and doesn't specify how the browser should implement those instructions. The benefits of declarative programming are that well-crafted declarative programs are concise and easily read, but they leave much of the work to the system — someone still had to use traditional programming worldviews and languages to create the functionality that performs the task.[363] In our HTML example, the browser is the software that implements the HTML instructions. However, HTML isn't a fully-fledged programming language. The best example of a declarative programming language is Prolog.[364]

All of these worldviews (object oriented programming, procedural programming, concurrent programming, declarative programming, and others) have become a part of the modern programmer's toolkit. Moreover, programmers often use parts of the different worldviews for different parts of the same software system. For example, a programmer may combine an object oriented worldview and a concurrent worldview to produce a large scale software system that runs on a computer with parallel hardware. This coexistence of worldviews is not unique to the computing field — in

---

[363] Markus Egger, "Anything To Declare?," *Code Magazine*, 1993, http://www.code-magazine.com/Article.aspx?quickid=050053, Accessed November 18, 2010.

[364] K.R. Apt, "Declarative Programming in Prolog," in *Procedures of the International Logic Programming Symposium (ILPS '93),* (Cambridge: MIT Press, 1993), 12-35.

physics, Newtonian mechanics has become a special case within Einstein's theory of relativity. However, unlike most sciences, currently there is little momentum towards a unified theory of computing. How this theme of plurality will play out in computing is an open question for future historians.

Plurality of Methods

Simultaneous with the increasing number of worldviews in programming, there was also an increase in the number of design methods. After the introduction of stepwise and top down design (described in chapter 4) more design methods appeared in the 1970s and early 1980s. The iterative method, created in response to criticisms of the waterfall method, is the best known of these early design methods.

The waterfall method originated in construction and engineering and was the de facto standard in software design prior to the structured programming controversy. It is a sequential development process, in which development flows through conception, initiation, analysis, design, construction, testing and maintenance. This method has been regularly altered to attempt to compensate for the many criticisms leveled at it — primarily, that software is never developed in discrete steps and that there must be an iterative interaction between

the steps. Despite this (valid) criticism, the waterfall method is still widely taught to students and used in commercial software development.

Alternatively, iterative design was created in response to criticisms of the waterfall method. It uses a sequence of repeated cycles. The project begins with initialization, where a basic version of the system is produced. This basic version is then iteratively improved with testing and additional design features. A project control list is used to keep track of the testing and features necessary to the final project.[365]

From these early design principles there has been a veritable explosion of design and development methods in recent years. These range from the chaotic agile development method to the increasingly formal Structured Systems Analysis and Design Methodology (SSADM).

The agile development method is said to promote teamwork, collaboration, and process adaptability throughout the life-cycle of the project. The method breaks tasks into small increments with minimal

---

[365] Craig Larman and Victor R. Basili, "Iterative and Incremental developments: A Brief History," *Computer* 36, no. 6 (2003): 47- 56

planning and without any detailed long term planning.[366] The agile development method has created numerous offshoots, including extreme programming and the scrum development method. Below is a diagram illustrating the agile software development method:

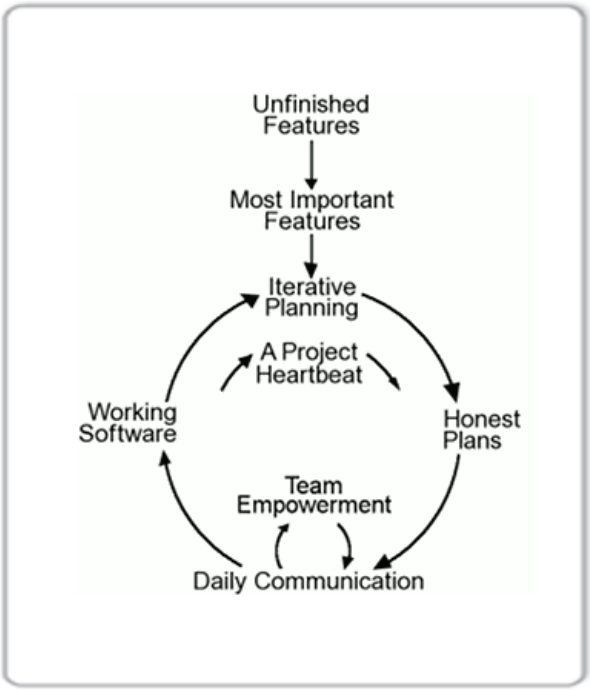


Figure 30: Agile Software Development

Source: http://www.agile-process.org/

Alternatively, the SSADM method is one of the many modified waterfall models. Created in the United Kingdom for the Office of Government Commerce, the method has three separate modeling phases, the logical data modeling phase, the data flow modeling

[366] P. Abrahamsson, J. Warsta, M.R. Siponen, and J. Ronkainen, "New Directions on Agile Methods: A Comparative Analysis," in *Proceedings of the 25th International Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003)*, (Washington D.C.: IEEE, 2003), 244-254.

phase, and entity behavior modeling. SSADM is a far more formal development process than the agile development method. Below is a diagram explaining the method:

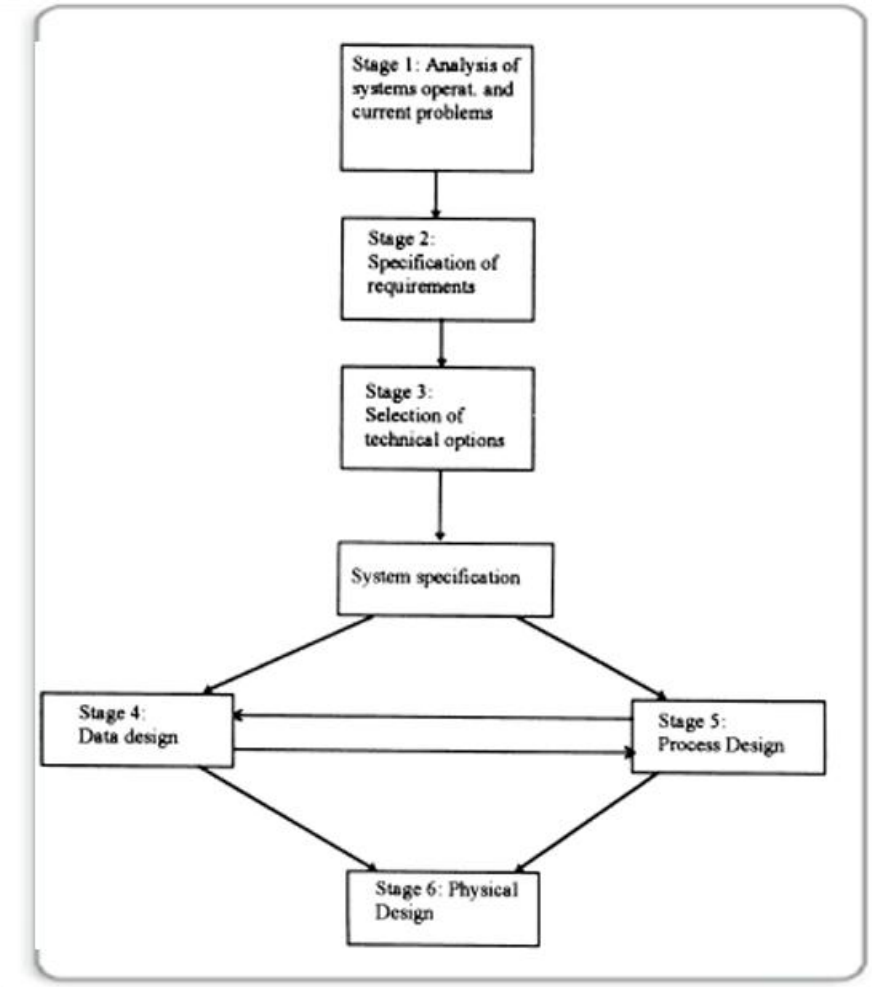


Figure 31: SSADM
Source: K. Vinodrai Pandya et, al, "Towards the manufacturing enterprises of the future", International Journal of Operations & Production Management, Vol. 17(5) [367]

[367] K. Vinodrai Pandya, Andreas Karlsson, Stefano Sega, and A. Carrie, "Towards the Manufacturing Enterprises of the Future," *International Journal of Operations & Production Management* 17, no. 5 (1997): 502 – 521.

These new design principles are not specific to a single programming worldview. However, they have been adopted by specific areas of application. For example, the principals of agile software development are frequently applied to web applications, while modified waterfall models (like SSADM) are often applied to more traditional, offline business applications.

A Reflection of Shared History

This increasing number of both programming worldviews and design methods is of particular relevance to this dissertation, because I argue that both changes are reflections of differing aspects of the history of software. I argue that the increase in the differing worldviews is a consequence of the adoption of object oriented programming. Alternatively, the rise of the many different design methods stems from the structured programming debate.

Object oriented programming was a significant change in the way programmers perceive programs, but this shift was able to coexist with the procedural (or process driven) programming worldview — the paradigm that was in place prior to the adoption of object oriented programming. Programmers were able to use the different worldviews to address different problem sets, depending on the best fit. This coexistence of procedural and object oriented programming opened

the floodgates for the adoption of new programming worldviews (like the concurrent and declarative programming styles) depending on which best suited the problem set on hand.

I argue that the plurality of design methods began earlier, with the structured programming debate. Prior to the structured programming debate, the waterfall method (although not called that until much later) was the de facto standard for program design. The structured programming debate introduced Dijkstra's stepwise design methods and Mills' top down approach. The vocational programming community used an integrated version of the two methods, as well as using each method independently, to solve different kinds of problems. The controversy over structured programming created a space for many different design methods, a trend that has continued after the culmination of the controversy. This resulted in the use of a plethora of different design methods by programming specialists and vocational programmers, alike. Having shown that structured programming was not the paradigm shift that its supporters claimed it to be, I argue that the plurality of methods in modern programming may be the one truly lasting legacy of the structured programming controversy.

Plurality of Languages

In addition to the plurality of worldviews and design methods, there has also been a tremendous increase in the number of programming languages since the mid-1980s. The rise of dynamic languages (languages like Java, Ruby, and Python) began immediately after the widespread adoption of C++.[368] The rise of dynamic languages confirms how pervasive the object oriented paradigm was in programming theory.

Java was created in the early 1990s, but there was little fanfare over its object oriented methodology. Instead, what made Java so interesting was that it was system independent. This easy acceptance of Java's object oriented methodology was a benefit Java reaped because of the by then widespread use of C++ and other object oriented languages. This widespread usage in other languages justified the use of an object oriented methodology in language design, generally. By the early 2000s, when Gosling was debating the merits of dynamic (scripting) languages, the war over object oriented programming had already been won. As I illustrated in chapter 5, many of the most popular programs used by the general public are

[368] Dynamic programming language is a term used broadly in computer science to describe a class of high-level programming languages that execute at runtime many common behaviors that other languages might perform during compilation.

written in C++, including operating systems, applications, databases, and games.

Furthermore, like design methods, languages have become not specific to a worldview but to areas of application, demonstrating a general trend in software towards an interchangeability of worldviews, with methods and languages clustered around areas of application. For example, the Java and ColdFusion languages are often applied to internet applications, while Actionscript (the Adobe Flash language) is used primarily for animation.

The plurality of languages that sprang up after the adoption of object oriented programming is significant to this dissertation for several reasons. The easy acceptance of the object oriented features of languages like Java and Python illustrate the pervasiveness of the object oriented paradigm. Moreover, the fact that many of the languages arising after the object oriented worldview were dynamic suggests that future explanations of change in programming practices must take the need for languages that can run on a variety of machines into account. Perhaps in future work, this type of universality of execution will be the kind of stimuli that verification and complexity have been between 1950s and the mid-1980s.

The New Landscape

Since the adoption of object oriented programming, the make-up of the computing field in both hardware and software has changed drastically. With the advent of the pc and the widespread adoption of the internet, distributed, networked computing has become the standard model of computing. This is reflected in the number of methods for communicating across systems. The increasing adoption of APIs (application program interfaces) in the early 1990s illustrates this need for communication between systems.[369] APIs extend the principles of data hiding, allowing external programs and programming languages access to the internal functions of the language or application through an interface. For example, applications that run on the Windows operating system access the file structure of the Windows operating system through APIs.[370]

Another example of this new need for flexibility across systems can be seen in Microsoft's BizTalk system, released in 2000. BizTalk facilitates and integrates XML-based business processes between

---

[369] I ascertained this increasing popularity through a literature search of IEEE. Please note, there is some confusion in the literature over IBM's APL (Application Programming Language) and APIs (Application Program Interface). There is an early article that refers to APIs in the description, but it is an artifact of optical character recognition. It appears this error has been carried over into a number of online sources, through a mistake in the OED.

[370] David Orenstein, "QuickStudy: Application Programming Interface (API)," *Computerworld,* January 10, 2000. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=43487 Accessed October 10, 2010.

organizations. XML (Extensible Markup Language) is a set of rules for encoding documents in machine-readable form. The BizTalk framework allows an organization to receive an XML document and parse the document based on specific schemas, and then integrate the documents into the proper applications. BizTalk worked using a pipeline between the organizations (or departments) that specifies inbound and outbound requirements of documents, which are enforced by adaptor technology. In this way, documents can be automatically shared between businesses and within large scale organizations.[371]

Yet just because new, dynamic systems have been developed, legacy systems don't simply disappear. Legacy systems were written in a different time, and they reflect the emphasis towards a universal solution that was popular during their time of inception. These legacy systems often lock the organizations they serve into rigid manners of using information.

A 2008 controversy in California illustrates this rigidity. In 2008 Governor Arnold Schwarzenegger, in a bid to balance the budget, recommended that temporary pay cuts be instated for approximately 200,000 hourly state workers and their managers. However, the state

---

[371] He Ning, Z. Milosevic, "B2B Contract Implementation using Windows DNS," in *Proceedings from the Workshop on Information Technology for Virtual Enterprises*, Volume 13 of *ITVE* (Washington D.C.: IEEE, 2003), 71-79.

controller John Chiang refused. Chiang argued that the pay cuts just weren't possible with the existing COBOL payroll accounting software. To complete the changes to the system would take at least six months, with a further 10 month waiting period to reinstate the discounted worker's payment.[372] Amazingly, while the 2008 case was still making its way through the court system, the situation arose again in 2010 and it, too, is still in litigation.[373] The conclusion of this story is still unknown, both because it is still in litigation and because of the overtly political nature of the debate, but the controversy is a very public example of the inflexibility of some legacy systems.

Moreover, there is always the difficulty for finding people to work on legacy systems. In an ironic twist to the Californian COBOL controversy, Schwarzenegger had fired part time COBOL programmers in the same round of budget cuts that required the temporary pay cut.[374] Even under less trying circumstances it is difficult to find COBOL programmers. Many COBOL programmers have retired and it

---

[372] Robert Charette, "COBOL Confounds California," *The Risk Factor: IEEE Spectrum Blog*, August 12, 2008. http://spectrum.ieee.org/riskfactor/computing/it/cobol_confounds_california Accessed October 4, 2010.

[373]Robert Charette, "Déjà Vu All Over Again: California Budget Cuts Stymied By Old IT Systems," *The Risk Factor: IEEE Spectrum Blog*, July 06, 2010. http://spectrum.ieee.org/riskfactor/computing/it/dj-vu-all-over-again-california-budget-cuts-stymied-by-old-it-systems Accessed October 4, 2010.

[374]Phil Machester, "COBOL Thwarts California's Governator: The Language that Refused to Die," *The Register*, August 14, 2008. http://www.theregister.co.uk/2008/08/14/cobol_california/ Accessed October 4, 2010.

is a skill no longer taught in traditional computer science departments. As Dan Tynan from *Infoworld* said in 2008, legacy system programming is one of the "dirtiest" jobs in computing.[375]

The ever more popular open source software movement is a counter to the rigidity of legacy systems. While legacy systems were written with a rigidity that reflected the large, inflexible institutions and corporations that needed these expensive new machines, the open source software movement reflects the touted characteristics of the internet — a distributed, networked, flexible community.

Open source software is software that is distributed with its source code. Source code is the human readable, symbolic code that can then be compiled or assembled into the executable code (machine language).[376] The benefit of providing source code is that users can manipulate the source code to meet their needs, re-compile or assemble the program into machine language and then use their newly altered application. Of course, distributing source code is not a new concept. In the early days of computing, all software was distributed with its source code because all software was customized at some

---

[375]Dan Tynan, "The 7 dirtiest jobs in IT," *InfoWorld*, March 10, 2008. http://www.infoworld.com/d/adventures-in-it/7-dirtiest-jobs-in-it-937 Accessed October 23, 2010.

[376] There are many moves towards more rigorous definitions of open source that encompass the everything from the development method to the legal terms of distribution, but for the purposes of this dissertation a simplistic definition is satisfactory.

point during its lifecycle. However, after IBM's unbundling decision, and as software became a commodity with the emergence of "shrink wrapped" software, most software providers chose to only distribute the executable code (the compiled machine language).[377]

The inception of the current open source software movement began with GNU, a UNIX compatible operating system, created by Richard Stallman in 1984. His motivation was to recreate the heyday of hacker culture he had found so stimulating during his time at MIT in the 1970s.[378] The open source software movement has grown significantly since these early days.[379]

Linus Torvalds' UNIX-like Linux operating system is one of the most well-known open source projects. In 1991, Linus Torvalds began working on a terminal emulator for his IBM 386. This emulator eventually became the kernel of the Linux operating system. Early into this process Torvalds described his work to the computing community and requested feedback about desired features. Linux

---

[377]Chris DiBona, Sam Ockman, and Mark Stone, "Introduction," in *Open Sources: Voices from the Open Source Revolution*, (Sebastopol: O'Reilly & Associates, 1999).

[378]Richard Stallman, "The GNU Operating System and the Free Software Movement," in *Open Sources: Voices from the Open Source Revolution*.

[379] Note: Stallman actually founded the "Free Software" movement, not the open source movement. While the free software movement isn't advocating directly for free software in the sense of price, following their argument results in cost-free software. The movement is far more radical than the open software movement, with openly ethical judgments on the nature of commercial software. Soon there was a split within the community, from which the open source software movement emerged.

was, from the beginning, intended to be an open source development.[380]

Mozilla, another well-known open source project, has a more storied beginning. The Mozilla story begins with Netscape Communications. In the 1990s Netscape's web browser, Navigator, was very popular. After Microsoft's 1995 release of Internet Explorer, Navigator was one of the few browsers threatening the hegemony of Internet Explorer on IBM-compatible PCs.[381]

In 1998 Netscape chose to freely release the source code for their popular Netscape browser. However, the Navigator tool had a number of third party applications embedded within the application. As a result, the application first had to be stripped of all the third party applications. Netscape created Mozilla.org[382] to manage this redevelopment: releasing the source code and the re-development of the application.[383] Currently, the browser released by the Mozilla

---

[380]Linus Torvalds, "The Linux Edge," in *Open Sources: Voices from the Open Source Revolution*.

[381] Mosaic was one of the only other browsers threatening IE's hegemony. Mosaic was developed at the University of Illinois Urbana-Champaign in 1992 and, combined with the University of Minnesota's Gopher system, is credited with popularizing the World Wide Web.

[382] Mozilla was the code name for the Netscape project and is used now to refer to the source code for the browser.

[383] Jim Hamerly and Tom Paquin with Susan Walton, "Freeing the Source: The Story of Mozilla," in *Open Sources: Voices from the Open Source Revolution.*

organization is Firefox.[384] After Netscape refocused on the open source project, the original Navigator software was sold to AOL (America Online), who continued to release it as traditional software package without source code.

The Mozilla project was wildly successful. The Firefox browser gained a 20% market share by 2008. To put this in perspective, in 2002, the year prior to the original Firefox release, Internet Explorer had a 90% share of the market. Mozilla source code has been used to develop many browsers, including AOL's Netscape 7 release, Camino (an Apple browser), and the SeaMonkey suite of web tools.[385]

As the Open Source Initiative states, the promise of open source is better quality software that is more reliable and flexible than traditional proprietary software.[386] Whether this comes to pass is still up for debate, but the popularity of open source software is intimately related to its flexibility. In a survey of 500 computing professionals, 66% used open source software. More interestingly for our purposes,

---

[384] Publicly releasing the Netscape source code affected more than just the Netscape Navigator application. The application called many third party provided modules, and each provider needed to be contacted and give consent to having their code publicly released, or Netscape programmers would need to replace the code. Many providers were enthusiastic, but at the time Java was Sun Microsystems' proprietary language. Since then, Java's source code has also been freely distributed, but at the time that was not an option. As a result, Netscape had to clean all of the heavily entwined Java code from their release.

[385] "History of the Mozilla Project," Mozilla Foundation, http://www.mozilla.org/about/history.html Accessed October 22, 2010.

[386] "Mission," Open Source Initiative Corporation, http://www.opensource.org/ Accessed October 22, 2010.

however, is that of those 66%, 70% of those respondents stated that the reason they used open source software was because of the flexibility it afforded them.[387]

Conclusion

This dissertation illustrated the rapid change in the computing field over its first fifty years: from the early days of mainframe, batch processing computing through the period of interactive, real time, time-shared mainframes and mini-computers, to the current conception of computing as distributed, networked, and personal. The focus of this dissertation demonstrates that concurrent with these changes in hardware, there have been equally important changes in programming theory and practice. At the beginning of this work, we explored the origin of programming, when it was conducted exclusively in machine language. We have seen the rise first of symbolic assembly languages and then higher level automatic coding systems. The design process moved from the ad hoc, flowcharted systems that followed the logic of the program, to modular structured design, and finally, object oriented programming — a sweeping change in worldview that prioritized objects over processes. Yet, complexity and

---

[387] Michael Vizard, "Flexibility Drives Open Source Adoption," *CTO Edge*, August 13, 2010. http://www.ctoedge.com/content/flexibility-drives-open-source-adoption Accessed October 22, 2010.

verification are still abiding topics and pervasive concerns in the software field.

The traditional narrative in computing has portrayed this as a story of technological progress, without analyzing why these changes to the method of writing software were instituted and how they were adopted by the community. This dissertation illustrates that the methodological changes in programming and new programming languages have been attempts to solve longstanding problems faced by programmers. Complexity and verification are two of the stimuli that acted as catalysts for these changes. Moreover, complexity and verification are not merely catalysts. By tracing the literature and comparing it with the changes made in the source code itself, complexity and verification have an explanatory power to explain the specific technical changes that occurred in program design and implementation.

Finding a method to address the questions I was interested in exploring was, by far, the most challenging element of this dissertation. From the inception of this project I intended to illustrate the technical changes in the design and implementation of software applications over time. I knew that I wanted to use actual source code to demonstrate technical change, not just rely on the rhetorical

changes described in the literature. I chose to supplement the primary literature with source code because the rhetoric of the field emphasizes revolution and novelty. Tracing the changes in software design and implementation through the computing literature, without reference to the technical changes appearing in the programs, could create a rabbit warren of loose ends and minor changes that were originally heralded as revolutionary. Using the technical changes to supplement the literature clarified the real versus the rhetorical changes.

Finding a method of organizing the information in this dissertation was challenging. To organize this work chronologically would create an entirely different problem. It could result in the arbitrary exclusion of major changes because they don't logically fit in to work of the time period during which they occurred. More importantly for this dissertation was the problem of interweaving two stories that, when described together, create a complex narrative and a cast of characters rivaling a Dostoevsky novel. Attempting to describe structured programming and object oriented programming chronologically would have this result – developed in parallel, through different academic lineages, the two methods have commonalities, but they are both pivotal developments with significant histories of their own.

The most challenging aspect of this dissertation has also been the most rewarding. Through this methodology a number of new directions of study have been exposed. While I have focused on two stimuli, complexity and verification, these are, undoubtedly, not the only stimuli that have been catalysts for change in program design and implementation. It is often claimed in the computing literature that change was driven by the quest for efficiency, a concept with its own rich and complex history. Portability (the range of hardware and operating system platforms on which the application can be executed), usability (the ease that a person can use the program), and maintainability (the ease with which a program can be modified) all appear throughout the literature as stimuli for change. Perhaps future research will assess if these stimuli have the same explanatory power to explain the technical changes in software design and implementation that complexity and verification have demonstrated.

This dissertation has also, unexpectedly, illustrated dramatic parallels between the structure of the hardware and their associated programming methodologies. In the early years of ad hoc, fragmented programming, the computers themselves were fragmented, pieced together from a variety of components, often only a little more sturdy than an experimental tool. During the rise of modular hardware (seen through the rise of the System/360, time

sharing, multiprogramming, and the mini-computer craze) we have the modular, structured programming methodology. In the distributed, networked personal computing phase, we have messaging based, object oriented programming. The seeming parallels between the structure of hardware and the software methodologies in computing are a fascinating conclusion from this dissertation, one that was unexpected at the outset of the project. Are these parallels just a correlation or is there some method of causation at work? Is this another instance of the confluence of ideas that we saw in the development of object oriented programming? While this research does not answer this question, it provides tantalizing glimpses to pursue in future work.

The history of software is a rich tapestry, not, as the internal history of the field suggests, a linear progression towards an ideal programming methodology and language. Nor is the history of software a Kuhnian tale of puzzle solving, crisis, and revolution — another historical narrative developed by participants in the computing field. Instead, I have presented a more nuanced interpretation of the history of software, by approaching technical change as being the result of problem solving activities. I have focused on two problems that have acted as stimuli for change: complexity and verification. These two problems not only influenced the direction of the field

between the 1950s and the 1980s, but continue to be a significant topic of research and debate in programming theory today.

Where will software go from here? It is impossible to predict. As James Noble and Robert Biddle illustrate in a tongue- in-cheek article for the ACM, the field seems to be embracing an increasingly instrumentalist attitude:

> "This postmodern programming stuff just seems to be an excuse to create software by sticking stuff together. Yeah, there doesn't really seem to be any theory at all…Or there's a whole bunch of theories. How could that ever work?"[388]

But, perhaps programming has always been a pragmatic activity and it is the people, captivated by its novelty, that try and elevate it to something more. In the 1982 film, *TRON*, the protagonist and the villain are both programmers who became their respective programs in order to wage an epic battle. The climactic scene of the film resolves this tension:

> "Stark: There's nothing special about you. You're just an ordinary program…
>
> Tron: So are you, one that should have been erased." [389]

---

[388] J. Noble and R. Biddle, "Notes on Notes on Postmodern Programming, *SIGPLAN Notice* 39, no. 12 (2004): 47.

[389] *TRON*, DVD, directed by Steven Lisberger, 1982, (Los Angeles, California, Walt Disney Pictures, 2004).

Works Cited

Abrahamsson, P., J. Warsta, M.R. Siponen, and J. Ronkainen. "New Directions on Agile Methods: A Comparative Analysis." In *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon May 03-10, 2003*: Washington D.C., IEEE. 2003. 244-254.

Adam, Sam. "The Future of End User Programming." In *Proceedings of the Thirtieth International Conference on Software Engineering, May 10-18, 2008,* edited by Wilhelm Schäfer, Matthew B. Dwyer, Volker Gruhn: Leipzig, Companion Volume ACM. 2008.

Adams, Eldridge S. "Simple Automatic Coding Systems." *Communications of the ACM* 1.7 1958: 5-9.

Akera, Atushi and Mitchell Marcus. "Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture." *IEEE Annals of the History of Computing* 18.1 1996: 17-24

Anthes, G. *COBOL Coders: Going, Going, Gone*?. 9 Oct. 2006. *Computerworld*. n.d. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=266228

Apt, K.R. "Declarative Programming in Prolog." Proceedings of the International Logic Programming Symposium (ILPS 93). Cambridge, MIT Press. 1993: 23-25 (http://homepages.cwi.nl/~apt/ps/a-mit.pdf)

Armstrong, Deborah J. "The Quarks of Object-Oriented Development." *Communications of the ACM* 49.2 2006: 123-128.

Backus et al. "The FORTRAN Automatic Coding System." *Proceedings of the Western Joint Computer Conference.* New York, Institute for Radio Engineers. 1957.

Backus, John. "The History of FORTRAN I, II and III." In *HOPL-I: Proceedings of History of Programming Languages – I, 1981*, Richard L. Wexelblat: Blue Bell: Sperry Univac. 1981. 25.

Backus, John. Personal interview. 5 Sept. 2006. Interview with Grady Booch. Computer History Museum, Ref. No X3715.2007: 8.

Bell, C.G. et. al. "A New Architecture for Mini-Computers – The DEC PDP-11." In *AFIPS Conference Proceedings 1970 Spring Joint Computer Conference, May 5-7, 1970*. New Jersey, AFIPS Press. 1970. 657-675.

Bell, C.G., C. Mudge, and J. McNamara. *Computer Engineering: A DEC View of Computer Design*. Bedford: Digital Press, 1978.

Bemer, R.W. "The View of the History of COBOL." Honeywell Computer Journal 5:3 1971: 133.

Bennington, Herbert D. "Production of Large Computer Programs." *IEEE Annals of the History of Computing* 5.4 1983: 357-358.

Bergin, Thomas J., and Richard G. Gibson, Jr., eds. *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993*. New York: ACM, 1994.

Beyer, Kurt. "Grace Hopper and the Early History of Computer Programming, 1944-1960." Diss. University of California at Berkeley, 2002.

Bohm, Corrado and Jacopini Giuseppe. "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules." *Communications of the ACM* 9.5 1966: 366-371.

Brandel, M. *The Top 10 Dead (or Dying) Computer Skills*. 24 May 2007. Computerworld. n.d. <http://www.computerworld.com/action/article.do?command=printArticleBasic&articleId=9020942>.

Bright, H.S. *Computers and Automation* 20.11 1971: 17-18.

Brooks, Jr., Fred. *The Mythical Man-Month: Essays on Software Engineering*. Reading: Addison-Wesley, 1975.

Campbell-Kelly, Martin. "Foundations of Computer Programming in Britain." Diss., Sunderland Polytechnic, 1980.

Campbell-Kelly, Martin. "Programming the EDSAC: Early Programming Activity at the University of Cambridge." *IEEE Annals of the History of Computing* 2.1 1980: 7

Campbell-Kelly, Martin. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge: MIT Press, 2004.

Campbell-Kelly, Martin and William Aspray. *Computer: A History of the Information Machine* 2nd Ed. Boulder: Westview Press, 2004.

Capretz, L. F. "A Brief History of the Object-Oriented Approach." *ACM SIGSOFT Software Engineering Notes* 28, no. 2 (2003): 1-10.

Cardelli, Luca and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism." *ACM Computing Surveys* 17.4 1985: 471-522

Carlson, William E. et. al. "Introducing Ada." In *ACM '80: Proceedings of the ACM 1980 Conference, January 1980.* New York, ACM. 1980.

Carpetz, L.F. *Software Engineering Notes* 28.2 2003: 3.

Ceruzzi, Paul E. *Reckoners*. Westport: Greenwood Press, 1983.

Ceruzzi, Paul E. “Crossing the Divide: Architectural Issues and the Emergence of the Stored Program Computer, 1935-1955.” *IEEE Annals of the History of Computing* 19.1 1997: 6.

Charette, Robert N. "Why Software Fails: We Waste Billions of Dollars Each Year on Entirely Preventable Mistakes." *IEEE Spectrum*, 2005.

Charette, Robert. “COBOL Confounds California.” *The Risk Factor: IEEE Spectrum Blog*, August 12, 2008. http://spectrum.ieee.org/riskfactor/computing/it/cobol_confounds_califgornia

Charette, Robert. “Déjà vu All Over Again: California Budget Cuts Stymied by Old IT Systems.” *The Risk Factor: IEEE Spectrum Blog*, July 6, 2010. http://spectrum.ieee.org/riskfactor/computing/it/dj-vu-all-over-again-california-budget-cuts-stymied-by-old-it-systems

Chivers, Ian D and Malcolm W. Clark. “History and Future of FORTRAN”, *Data Processing*, 27.1 1985: 39 – 41.

Clark, Lawrence R. “We Don’t Know Where to GOTO if We Don’t Know Where We’ve COME FROM. This Linguistic Innovation Lives Up to All Expectations.” *Datamation* Dec. 1973 http://www.fortran.com/come_from.html

Collins, H.M. "The TEA Set: Tacit Knowledge and Scientific Networks." *Science Studies* 4, no. 2 (1974): 165-85.

Constantine, et. al.,. "Structured Design." *Tutorial on Structured Programming: Integrated Practices.* Ed. V.R. Basili and F.T. Baker. Washington D.C.: IEEE Computer Society Press, 1981. 148-152.

Copeland, Duncan G., Richard Hanson and James L. McKenney, "Sabre: The Development of Information-Based Competence and Execution of Information-Based Competition." *IEEE Annals of the History of Computing* 17.3 1995: 30-33.

Cortada, James W. *Historical Dictionary of Data Processing Technology*. New York: Greenwood Press, 1987.

Dahl, et. al. "Some Features of the Simula 67 Language." In *Winter Simulation Conference: Proceedings of the Second Conference on Applications of Simulations, December 1968.* New York, Winter Simulation Conference. 1968.

Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. New York: ACM Classic Books, 1972.

Denning, Peter J. "Third Generation Computer Systems." *ACM Computer Systems* 3.4 1971: 175 - 216

Denning, Peter J. "Is it Not Time to Define 'Structured Programming'." *ACM SIGOPS Operating Systems Review* 8.1 1974: 6-7

Denning, Peter J. "Two Misconceptions about Structured Programming." In *ACM 75: Proceedings of the 1975 Annual Conference, 1975.* New York, ACM. 1975. 214.

Denning, Peter J. "Computer Science: The Discipline." In *Encyclopedia of Computer Science*, eds. A Ralston and D. Hemmendinger. Chichester: John Wiley and Sons Ltd., 2003: 405-419.

Deutsch and Shea, Inc. "A Profile of the Programmer." *Industrial Relations News, Inc.* 1963: 1-2.

Dictionary and Thesaurus - Merriam-Webster Online. Accessed November 23, 2011. http://merriam-webster.com.

DiBona, Chris, Sam Ockman, and Mark Stone. "Introduction." *Open Sources: Voices from the Open Source Revolution*. Sebastopol: O'Reilly & Associates, 1999.

Dijkstra, Edsger W. "To Howard Aiken", _____, Edsger W. Dijkstra Papers, 1948-2002, Archives for American Mathematics, Center for American History, The University of Texas at Austin.

Dijkstra, Edsger, W. "Transcript of Some Meditations on Advanced Programming Presented at the IFIP Congress Munich," 1962, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Dijkstra, Edsger W. "Letter to the Editor: Go To Statement Considered Harmful." *Communications of the ACM* 11.3 1968: 147.

Dijkstra, Edsger W. personal correspondence,____, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin: 10

Dijkstra, Edsger W. "A Constructive Approach to the Problem of Program Correctness," 1967, EWD2090. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Dijkstra, Edsger W. "Towards Correct Programs," 1968, EWD241. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin

Dijkstra, Edsger W. "Notes on Structured Programming," 1969, EWD249. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin. 10.

Dijkstra, Edsger W., "Forty Years of Computer Science Devoted to Education," 1989, EWD1051-1. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin

Dijkstra, Edsger W., "Computing Science: Achievements and Challenges," 1989, EWD1284. Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Dijkstra, Edsger W. Privately Circulated Note, 2001, "What led to "Notes on Structured Programming?" EWD 1308, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Doernberg, Dan Interview with Donald Knuth, for the Computer Literacy Bookshop, December 1993, http://tex.loria.fr/litte/knuth-interview

Dolotta, T.A. *Data Processing in 1980-1985: A Study of Potential Limitations to Progress.* New York: John Wiley and Sons, 1976.

Durbin, Gary. Personal interview. 3 May 2002, OH 368. Oral history interview by Philip L. Frana, Washington, DC. Charles Babbage Institute, University of Minnesota, Minneapolis: 4.

Eckert–Mauchly Computer Corporation, "Future Design Group" memo, August 15, 1950, Frances E. Holberton Papers, Box 13, Charles Babbage Institute, University of Minnesota, Minneapolis.

Eckert–Mauchly Computer Corporation, Unpublished flowchart, "Floating Decimal Subroutine for Addition, Multiplication, Division," March 31, 1948, Frances E. Holberton Papers, Box 13, Charles Babbage Institute, University of Minnesota, Minneapolis.

Egger, Marcus. "Anything to Declare?" *Code Magazine*, 1993. http://www.code-magazine.com/Article.aspx?quickid=050053

Ensmenger, Nathan. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge: MIT Press, 2010

Fenichel, Robert R. "On Implementation of Label, Variables." *Communications of the ACM*14, no. 5 (1971): 349-50.

Glass, Robert L. "Cobol - A Contradiction and an Enigma", *Communications of the ACM*, 40.9 1997, 11-13

Hopper, Grace. OH 81. Oral history interview by Christopher Evans, 1976, Science Museum, convenience copy at the Charles Babbage Institute, University of Minnesota, Minneapolis

Fenichel, Robert R. "On Implementation of Label, Variables." *Communications of the ACM* 14.5 1971: 349-350.

Fetzer, James "Program Verification: The Very Idea." *Communications of the ACM* Vol. 31 Issue 9, 1988.

Fisher, I. and M. Bradford. "New York State Agencies: A Case Study for Analyzing the Process of Legacy System Migration, Part 1." *Journal of Information Systems* 19-2 2005: 173-189.

Fitzwater, Donald, and Earl Schweppe. "Consequent Procedures in Conventional Computers." In *Proceedings of AFIPS '64 Fall Joint Computer Conference, October 27-29, 1964.* New York, ACM. 1964. 1065.

Fletcher, John G. "No! High Level Languages Should Not be Used to Write Systems Software." In *Proceedings if the 1975 ACM Annual Conference.* New York: ACM, 1975.

Forrester, J. "Digital Computers: Present and Future Trends." In *Joint AIEE-IRE Computer Conference: Review of Electronic Digital Computers.* New York, American Institute of Electrical Engineers. 1952.

Galland, Frank J. Galland *Dictionary of Computing*. New York: John Wiley & Sons, 1982.

Goldstein, Harry. "Who Killed the Virtual Case File: How the FBI Blew More than 100 Million on Case-Management Software It Will Never Use." *IEEE Spectrum*, 2005.

Greenfield, Martin. "History of FORTRAN Standardization." In *AFIPS Conference Proceedings, 1982 National Computer Conference, June 7-10 1982*. Howard Lee Morgan: Arlington, AFIPS Press. 1982.

Grad, Burton. "A Personal Recollection: IBM's Unbundling of Software and Services." *IEEE Annuals of the History of Computing* 24.1 2002: 65.

Grier, David Allan. "The Eniac, the Verb "to Program" and the Emergence of Digital Computers." *IEEE Annals of the History of Computing* 18.1 1996: N. pag.

Gries, David. "On Structured Programming – A Reply to Smoliar." *Communications of the ACM* 17.11 1974: 655.

Gupta, G.K. "Computer Science Curriculum Developments in the 1960s." IEEE Annals of the History of Computing 29.2 2007: 45-52.

Haigh, Thomas. "Software in the 1960s as Concept, Service, and Product." *IEEE Annals in the History of Computing* 24.1 2002: 9.

Haigh, Thomas David. "Technology, Information, and Power: Managerial Technicians in Corporate America, 1917-2000." Diss., University of Pennsylvania, 2003.

Hamerly, Jim, Tom Paquin and Susan Walton. "Freeing the Source: The Story of Mozilla." *Open Sources: Voices from the Open Source Revolution*. Sebastopol: O'Reilly & Associates, 1999.

Hashagen, Ulf, Reinhard Keil-Slawik, and Arthur Norberg. *History of Computing: Software Issues*. Berlin: Springer, 2002.

Head, R.V. "Sabre Planning." Unpublished paper. 1962. Robert V. Head Papers (CBI 170). Charles Babbage Institute, University of Minnesota, Minneapolis.

History of the Mozilla Project. Accessed October 22, 2010. http://www.mozilla.org/about/history.html.

Hoare, C.A.R. *Hints on Programming Language Design*, Stanford University: Stanford, 1973

Hoare, C.A.R. "Monitors: An Operating System Structuring Concept." *Communications of the ACM* 17.10 1974: 549 - 557

Hoare, C.A.R. OH 357. Oral history interview by Philip L. Frana, 17 July 2002, Cambridge, England, U.K. Charles Babbage Institute, University of Minnesota, Minneapolis.

Holberton, Frances E. "Future Desigh Group Memo." Frances E. Holberton Papers, Box 13. Charles Babbage Institute, University of Minnesota, Minneapolis.

Holberton, Frances E. "Unpublished Flowchart, Floating Decimal Subroutine for Addition, Multiplication, Division." Frances E.

Holberton Papers, Box 13. Charles Babbage Institute, University of Minnesota, Minneapolis.

Holberton, Frances E. Personal interview. 14 Apr. 1983, OH 50. Oral history interview by James Baker Ross, Potomac, Maryland. Charles Babbage Institute, University of Minnesota, Minneapolis.

Holberton, Frances E. Personal interview. 14 Apr. 1983, OH 81. Oral history interview by James Baker Ross, Potomac, Maryland. Charles Babbage Institute, University of Minnesota, Minneapolis.

Hopkins, Martin E. "A Case for the GOTO." In *ACM '72: Proceedings of the ACM Annual Conference – Volume 2*. New York: ACM, 1972.

Hopper, Grace. "Keynote Address." In *HOPL-I: Proceedings of History of Programming Languages - I, 1981.* Richard L. Wexelblat: Blue Bell, Sperry Univac. 1981. 8-11.

Hopper, Grace. Personal interview, 1976

Interview with Christopher Evans, Science Museum, 1976. Charles Babbage Institute, University of Minnesota, Minneapolis.

Hopper, Grace. Personal interview. Dec. 1980.

Interview with Angeline Pantages, Naval Data Automation Command, Maryland, December, 1980. (Computer History Museum, CHM Reference Number X5142.2009).

Hudson, A. R., S. H. Pazkad, and J. B. Cheng. "Object-Oriented Database Management Systems: Evolution and Performance Issues." *Computer* 26, no. 2 (1993): 48-60.

Hughes, Robert A. "Early FORTRAN at Livermore." *Special Issue of the IEEE Annals of the History of Computing* 6.1 1984: 30-31.

Hull, T.E.. "Would You Believe Structured FORTRAN." *ACM SIGNUM Newsletter* 8.4 1973: 13-16.

IBM. FORTRAN Monitor for the IBM 709 (manual), IBM, 1959-1960. (Computer History Museum, http://www.computerhistory.org/collections/accession/102678932

IBM, "Inventory File." Unpublished paper, 1959. Robert V. Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis.

Jacobs, John F. "SAGE Overview," *IEEE Annals of the History of Computing* 5. 4 1983: 323-329

Johnson, Doug et al. "Managing Object Oriented Projects." In *Proceedings of the Tenth Annual Conference on Object Oriented Programming Systems, Languages, and Applications, October 15-19, 1995.* New York, ACM. 1995.

Johnson, Luanne. "A View From the 1960s: How the Software Industry Began." *IEEE Annals of the History of Computing* 20.1 1998: 36.

Jones, C.B. "The Early Search for Tractable Ways of Reasoning About Programs." *IEEE Annals of the History of Computing*, Vol. 25 Issue 2, 2003

Jones, Douglas E. "A Programming Aid for Structured Programmers." In *Proceedings of the 1974 Annual ACM Conference Volume 2, January 1, 1974.* New York, ACM. 1974. 676.

Kay, Alan C. "The Early History of Smalltalk." In *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993.* Edited by Thomas J. Bergin, Jr. and Richard G. Gibson, Jr: New York, ACM. 1994.

King, James, and Robert Floyd. "An Interpretation Oriented Theorem Prover Over Integers." In *STOC '70: Proceedings of the Second Annual ACM Symposium on Theory of Computing.* New York, ACM. 1970. 169.

Kidder, Tracy. *The Soul of a New Machine*. Boston: Little, Brown & Company, 1981. 32-33

Larman, Craig and Victor R. Basili. "Iterative and Incremental Developments: A Brief History." *Computer* 36.6 2003 47-56.

Leavenworth, B.M., "Programming with(out) the GOTO", *Proceedings of the 1972 ACM Annual Conference, Vol. 2* New York: ACM, 1975.

Lee, J.A.N. and H.S. Tropp. "FORTRAN's Twenty-Fifth Anniversary." *Special Issue of the IEEE Annals of the History of Computing* 6.1 1984: 7-14.

Lee, J.A.N. "Pioneer Day." *IEEE Annals of the History of Computing* 6.1 1984: 13.

Llamosi, Albert, Pere Botella, and Fernando Orejas. "On Unlimited Types and Reliability of Ada Programs." *SIGAda Ada Letters* IV.1 1984: 50 – 60

Macalester, Phil. "COBOL Thwarts California's Governator: The Language that Refused to Die." *The Register*, August 14, 2008 http://www.theregister.co.uk/2008/08/14/cobol_california/

Mahoney, Michael S. "The History of Computing in the History of Technology." *Annals in the History of Computing*. 10 (1988): 113-125.

Mahoney, Michael S. "The Roots of Software Engineering." *CWI Quarterly*. 3.4 (1990): 327.

Mahoney, Michael S. "Issues in the History of Computing." *History of Computing: Software Issues* ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur Norberg. Berlin: Springer Verlag, 2002.

Mahoney, Michael S. "Software: The Self-Programming Machine." *From 0 to 1: An Authoritative History of Modern Computing.* Ed.

Atsushi Akera and Frederik Nebeker. New York: Oxford University Press, 2002.

Mahoney, Michael S. "Software as Science – Science as Software." *History of Computing: Software Issues* ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur Norberg. Berlin: Springer Verlag, 2002.

Mahoney, Michael S. "The Histories of Computing." *Interdisciplinary Science Reviews.* 30.2 (2005)

Mantel, Marilyn. "The Effect of Programming Team Structures on Programming Tasks." *Communications of the ACM* 24.3 1981: 106-113.

McCarthy, J. "Towards a Mathematical Science of Computation." In *Information Processing 1962: Proceedings of IFIP Congress 62.* Edited by C.M. Popplewell: Amsterdam, North Holland. 1963. 21-28.

McCarthy, John. "History of LISP." *SIGPLAN Notices* 13.8 1978: 217 - 223

McCarthy, John. "History of LISP." In *HOPL-I: Proceedings of History of Programming Languages - I, 1981.* Edited by Richard L. Wexelblat: Blue Bell, Sperry Univac. 1981. 173.

MacKenzie, Donald. “The Automation of Proof: A Historical and Sociological Exploration.” *IEEE Annals of the History of Computing* 17.3 1995: 7-29.

MacKenzie, Donald. *Knowing Machines*. Cambridge: MIT Press, 1998.

MacKenzie, Donald. "Computer Related Accidental Death." In *Knowing Machines*, edited by Donald MacKenzie, 185-214. Cambridge: MIT Press, 1998

MacKenzie, Donald. "Fangs of the VIPER." In *Knowing Machines*, edited by Donald MacKenzie, 158-64. Cambridge: MIT Press, 1998.

MacKenzie, Donald. "Negotiating Arithmetic, Constructing Proof." In *Knowing Machines*, edited by Donald MacKenzie, 178-183. Cambridge: MIT Press, 1998.

MacKenzie, Donald. "Tacit Knowledge and the Uninvention of Nuclear Weapons." In *Knowing Machines*, edited by Donald MacKenzie, 215. Cambridge: MIT Press, 1998.

MacKenzie, Donald. "Nuclear Weapons and Supercomputing" In *Knowing Machines*, edited by Donald MacKenzie, 125-129. Cambridge: MIT Press, 1998.

MacKenzie, Donald. *Mechanizing Proof: Computing, Risk, and Trust*. Cambridge: MIT Press, 2004.

Mantai, Marilyn. "The Effect of Programming Team Structures on Programming Tasks." *Communications of the ACM* 24.3 1981: 106-113.

Meissner, Loren P. "A Method to Expose the Hidden Structure of FORTRAN Programs." In *Proceedings of the 1974 Annual ACM Conference Volume 1, January 1, 1974.* New York, ACM. 1974. 193.

Meyer, Bertrand, "Interview with Programming Expert Bertrand Meyer," November 30, 2001, *InformIT* (courtesy of Prentice Hall PTR), http://se.ethz.ch/~meyer/publications/interviews/informit_interview.html, Accessed July 27, 2010

Mills, Harlan D. "How to Write Correct Programs and Know It." In *Proceedings of the International Conference on Reliable Software, 1975.* New York, ACM. 1975. 363-370.

Mills, Harlan D. "Mathematical Foundations for Structured Programming." *Tutorial on Structured Programming: Integrated Practices.* Ed. V.R. Basili and F.T. Baker. Washington D.C.: IEEE Computer Society Press, 1981. 50.

Mission, Open Source Initiative Corporation. Accessed October 22, 2010. www.opensource.org.

Moon, David A. "Object Oriented Programming with Flavors." *Proceedings of the 1986 Conference on Object Oriented Programming Systems, Languages, and Applications, 1986.* New York, ACM. 1986.

Mozilla Foundation. "History of the Mozilla Project." http://www.mozilla.org/about.history.html

Naur, Peter. personal correspondence, 1970, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Naur, Peter. personal correspondence, 1962, Edsger W. Dijkstra Papers, 1948-2002, Archives of American Mathematics, Center for American History, The University of Texas at Austin.

Naur, Peter. "Proof of Algorithms by General Snapshots." *BIT* 6. 1966: 310.

Ning, He and Z. Milosevic. "B2B Contract Implementation Using Windows DNS." *In Proceedings from the Workshop on Information Technology for Virtual Enterprises, Vol. 13 of ITVE.* Washington, D.C.: IEEE. 2003. 71-79.

Noble, J. and R. Biddle. "Notes on Notes on Postmodern Programming." *SIGPLAN Notices* 39.12: 2004: 47.

Nofre, D. "Unraveling Algol: US, Europe, and the Creation of a Programming Language." *IEEE Annals of the History of Computing* 32.2 2010: 58-68.

Norberg, Arthur and Jeffrey Yost. "IBM Rochester, A Half Century of Innovation." *IBM* 2007, 36-37.

Nygaard, Kristen and Ole-Johan Dahl. "Simula: An ALGOL-Based Simulation Language." *Communications of the ACM* 9.9 1966: 671 - 678

Nygaard, Kristen and Ole-Johan Dahl. "The Development of the Simula Languages." *SIGPLAN Notices* 13.8 1978: 245 - 272

Nygaard, Kristen and Ole-Johan Dahl, "The Development of the Simula Languages." In *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993*. Edited by Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (New York: ACM, 1994)

Office of the New York State Comptroller. http://osc.state.ny.us/agencies/cas/index.htm

Open Source Initiative Corporation. "Mission." http://www.opensource.org

Orchard-Hays, W. "A New Approach to the Programming Problem." In *Proceedings of the IRE-AIEE-ACM Western Joint Computer Conference, May 3-5 1960:* New York, ACM. 1960.

Orenstein, David. "QuickStudy: Application Programming Interface (API)." Computerworld, January 10, 2000 http://www.computerworld.com/s/article/43487/Application_Programming_Interface

Page, E.S., Gotlieb, C.C. "Business and management data processing II: On the scheduling of jobs by computer," *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers*: New York, ACM. 1962.

Pandya, K. Vinodrai, Andreas Karlsson, Stefano Sega, and A. Carrie. "Towards the Manufacturing Enterprises of the Future." *International Journal of Operations and Production Management* 17.5 1997 502-521.

Perlis, Alan. "The American Side of the Development of ALGOL." In *HOPL-I: Proceedings of History of Programming Languages – I, 1981.* Edited by Richard L. Wexelblat: Blue Bell, Sperry Univac. 1981. 90.

Pescio, Dr. Carlo. "A Few Words with Niklaus Wirth." *Software Development* 5.6 1997: 52 - 60

Priestley, Peter Mark. *Logic and the Development of Programming Languages, 1930 – 1975*. Diss., University College, London, 2008.

Pugh, Emerson. *Building IBM: Shaping Industry and its Technology*. Cambridge: MIT Press, 1995.

Pyle, I.C. and Valerie Illingsworth. *A Dictionary of Computing*, 4th ed. New York: Oxford University Press, 1997.

Randell, B. "The Origins of Computer Programming." *IEEE Annals of the History of Computing* 16.4 1994: 13.

Rice, John R. "Letters to the Editor: The Go To Statement Reconsidered." *Communications of the ACM* 11.8 1968: 538.

Rosen, Saul. "Programming Systems and Languages: A Historical Survey." In *Proceedings of AFIPS '64 Spring Joint Computer Conference, April 21-23,* New York:ACM, 1964.

Rothstein, M. "American Airlines 709-7090 Training Program." Unpublished document, April 6, 1960. Robert V. Head Papers (CBI 170), Charles Babbage Institute, University of Minnesota, Minneapolis.

Rosen, Saul. "Programming Systems and Languages: A Historical Survey." In *Proceedings of AFIPS '64 Spring Joint Computer Conference, April 21-23, 1964.* New York, ACM. 1964. 8-10.

Rubin, Frank. "'GOTO Considered Harmful' Considered Harmful." *Communications of the ACM* 30.3 1987: 195-196.

Ryder, Barbara G., Mary Lou Soffa and Margaret Burnett. "The Impact of Software Engineering Research on Modern Programming

Languages.” *Transactions on Software Engineering and Methodology (TOSEM)* 14.4

Rustan,K., M. Leino, Greg Nelson, “Data Abstraction and Information Hiding,” *Transactions on Programming Languages and Systems (TOPLAS)* Vol. 24 Issue 5, 2001

Ryder, Barbara G, Mary Lou Soffa, and Margaret Burnett. “The Impact of Software Enginnering Research on Modern Programming Languages.” *Transactions on Software Engineering and Methodology* 14.4, 2005, 431 - 477

Sammet, J.E. *Programming Languages: History and Fundamentals.* Englewood Cliffs: Prentice-Hall, 1969.

Sammet, Jean. “The Early History of COBOL.” In *HOPL-I: Proceedings of History of Programming Languages - I, 1981.* Edited by Richard L. Wexelblat: Blue Bell, Sperry Univac. 1981.

Sammet, J.E.. “The Real Creators of COBOL.” *IEEE Annals of the History of Computing* 17.2 2000: 30-32.

Schildt, Herbert. *C++ The Complete Reference Third Edition*. Osborne: McGraw-Hill, 1998.

Schmitt, W.F.. “Univac Short Code*.” IEEE Annals of the History of Computing* 10.1 1988: 11.

Shapiro, Stuart S. *Computer Software as Technology: An Examination of Technological Development*. Pittsburgh: Carnegie Mellon, 1990.

Shneiderman, Ben. "A Short History of Structured Flowcharts (Nassi-Shneiderman Diagrams)." http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/.

Shull, F., V. Basili, J. Carver, G.H. Travassos, M. Mendocana, and S. Fabbri. "Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem." In *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, 7-16. Washington D.C.: IEEE Computer Society, 2002.

Smoliar, S. "Letter to the Editor: On Structured Programming." *Communications of the ACM* 17.11 1974: 294.

Soare, R.I.. "Computability and Recursion." *Bulletin of Symbolic Logic 2* 1996: 284-321.

Stallman, Richard. "The GNU Operating System and the Free Software Movement." *Open Sources: Voices from the Open Source Revolution*. Sebastopol: O'Reilly & Associates, 1999.

Steel, Jr., T.B. "Automatic Programming and Compilers III: Languages and Real Time Information Processing." In *Proceedings of the*

1962 ACM National Conference on Digest of Technical Papers, 1962. New York, ACM. 1962. 90.

Steele., Jr., Guy L., and Richard P. Gabriel. "The Evolution of LISP." In *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, April 20-23, 1993.* Edited by Thomas J. Bergin, Jr. and Richard G. Gibson, Jr.: New York, ACM. 1994.

Stein, Marvin L and David A. Pope. *Coding the Modern Digital Computer – With Special Reference to Univac Scientific Model 1103.* Minneapolis: University of Minnesota, 1960.

Stern, Nancy B. *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers.* Bedford: Digital Press, 1981.

Stroustrup, B. (2011) *C++ Applications*. http://www2.research.att.com/~bs/applications.html.

Stroustrup, B. *The C++ Programming Language.* Murray Hill: AT&T Labs, 2000.

Stroustrup, Bjarne. "On Unifying Module Interfaces." *SIGOPS Operating Systems Review* 12.1, 1978: 90 - 98

Stroustrup, Bjarne. *The Design and Evolution of C++.* Reading: Addison Wesley, 1994.

Sutherland, Ivan E. "Sketchpad: A Man-Machine Graphical Communication System ." In *Proceedings of the AFIPS 1963 Spring Joint Computer Conference, May 21-23 1963.* Baltimore, Spartan Books. 1963.

Sutter, Herb. "The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling." *Java Report* 5.7 2000

"That was Funny," Accessed on July 27, 2010, http://www.thatwasfunny.com/programming-languages-are-like-cars/1509

Tomayko, James E. "Structured Design." *IEEE Annals of the History of Computing* 12.4 1990: 269-276.

Torgerrson, Olof. "A Note on Declarative Programming Paradigms and the Future of Definitional Programming." http://www.cse.chalmers.se/~oloft/Papers/wm96/wm96.html

Torvalds, Linus. "The Linux Edge." *Open Sources: Voices from the Open Source Revolution*. Sebastopol: O'Reilly & Associates, 1999.

Townsend, Eric S. "Wells Fargo's 'Object Express'." *Distributed Object Computing* 1.1 1997: 18-27.

Training Section of the Electronic Computing Department of Remington Rand. *Central Computer of the UNIVAC System, Manual of Operations.* 1954.

Tracz, Will. "The Real Stroustrup Interview." *Computer* 31.6 1998: 110-114

Tropp, Henry S. et. al., "A Perspective on Sage: Discussion." *IEEE Annals of the History of Computing* 5.4 1983: 380.

Tynan, Dan. "The 7 Dirtiest Jobs in IT." Infoworld, March 10, 2008. http://www.infoworld.com/d/adventures-in-it/7-dirtiest-jobs-in-it-937

Usselman, Steven W. "Unbundling IBM: Antitrust and the Incentives to Innovation in American Computing." In *The Challenge of Remaining Innovative: Insights from Twentieth-Century American Business*, eds. Sally H. Clarke, Naomi Lamoreaux, Steven W. Usselman. Stanford: Stanford University Press, 2009.

Usselman, Steven. W. "Public Policies, Private Platforms: Antitrust and American Computing." In *Information Technology Policy* ed. Richard C. Coopey. Oxford: Oxford University Press, 2004.

Vizard, Michael. "Flexibility Drives Open Source Adoption." *CTO Edge*, August 13, 2010. http://www.ctoedge.com/content/flexibility-drives-open-source-adoption

von Neumann, John. “The First Draft of a Report on the EDVAC.” *IEEE Annals of the History of Computing* 15.4 1994: 8.

Walt Disney Pictures. *TRON*. DVD Directed by Steven Lisberger. Los Angeles, 1982.

Wernick, P, and T. Hall. "Can Thomas Kuhn's Paradigms Help Us Understand Software Engineering?" *European Journal of Information Systems.* 13.3 (2004): 235-243

Wexelblat, Richard L., ed. *HOPL-I: Proceedings of History of Programming Languages - I*. Blue Bell: Sperry Univac, 1981.

Wilkes, Maurice. *Memoirs of a Computer Pioneer*. Cambridge: MIT Press, 1985.

Wilkes, Maurice V. “Computers Then and Now.” *Journal of the ACM* 15.1 1968: 1-7.

Winskel, Glynn and Mogens Nielsen. “Models for Concurrency.” In *The Handbook of Logic in Computer Science Vol 4: Semantic Modeling*, eds. S. Abramsky, Dov. M. Gabbay, and T.S.E. Maibum. Oxford: Oxford University Press, 1995: 1-148.

Wirth, Niklaus. "An Assessment of the Programming Language Pascal." In *Proceedings of the International Conference on Reliable Software, April 21-23 1975.* New York, ACM. 1975. 23-30.

Wirth, Niklaus. "Recollections about the Development of Pascal." In *HOPL-II: Proceedings of the Second ACM SIGPLAN Conference*

on History of Programming Languages, April 20-23, 1993. Edited by Thomas J. Bergin, Jr. and Richard G. Gibson, Jr.: New York, ACM. 1994. 97.

Wirth, Niklaus. "Computing Science Education: The Road Not Taken." In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, June 24-28, 2002.* New York, ACM. 2002.

Wirth, Niklaus. "Modular-2 and Oberon." *HOPL-III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, June 9-10, 2007.* New York, ACM. 2007.

Wyatt, Sally. "Determinism is Dead; Long Live Technological Determinism." *The Handbook of Science and Technology Studies,* 3rd. ed.. Ed. Edward J.Hackett et. al. Cambridge: MIT Press, 2008

Xia, F. "Software Engineering Research: A Methodological Analysis." In *Fourth Asia-Pacific Software Engineering and International Computer Science Conference.* (1997): 229-236.

Yost, Jeffrey R. *The Computer Industry*. New York: Greenwood Press, 2005.

Appendix One

Original Research

Segment of Donald Knuth's TPK Algorithm written in A2

1	2	3	4	5	6	7	8	9	10	11	12	Notes
G	**M**	**I**	**0**	**2**	**A**	**0**	**0**	**1**	**0**	**0**	**0**	Specify the tape output & the number of words to be transferred from the input tape
Q	**T**	**A**	**0**	**0**	**8**	**0**	**0**	**0**	**0**	**0**	**0**	Test that loc 000 is greater than 0
A	**S**	**0**	**0**	**0**	**8**	**0**	**0**	**8**	**0**	**1**	**2**	Subtract loc 8 from loc 8 place the result in loc 12
A	**S**	**0**	**0**	**0**	**8**	**0**	**0**	**8**	**0**	**1**	**2**	Subtract location 8 from loc 12 place the result in loc 12

Segment of Donald Knuth's TPK Algorithm written in FORTRAN

```
INTEGER A,C
READ(*,*) A, B
IF(A.LT 0) STOP 1
B = (A – A)
STOP
END
```

Appendix Two
Original Research

Below is pseudo code for a flight search program, first in the style of structured programming and then in the style of ad hoc coding of the 1960s.

```
FlightSearch Program
{
        /*Start*/
        /*variable set up etc.*/
        Call (Subroutine LoginToFlightDatabase)
                If Subroutine LoginToFlightDatabase returns True
                        {
                                Call (Subroutine SearchFlightDatabaseCity(Parameter CityA,
                                Parameter CityB))
                        }
                Else
                        {
                                Call(Subroutine LoginError)
                        }
                        If  Subroutine SearchFlightDatabaseCity returned > 1
                                {
                                        Call (Subroutine SearchFlightDatabaseTime)
                                }
                        Else If Subroutine SearchFlightDatabaseCity returned = 1
                                {
                                        Call (Subroutine DisplayFlights(Parameter FlightList))
                                }
                        Else if Subroutine SearchFlightDatabaseCity returned = 0
                                {
                                Call Subroutine (FlightsError)
                                }
                If Subroutine SearchFlightDatabaseTime(Parameter FlightList) returned > 0
                        {
                        Call (Subroutine DisplayFlights(Parameter FlightList))
                        }
                If Subroutine SearchFlightDatabaseTime returned < 1
                        {
                        Call (Subroutine FlightsError)
```

```
                    }
/*End Process*/
/*Define Subroutines*/


Subroutine SearchFlightDatabaseLogin
        {
        /*Function that logs in registered users of the Database*/
        }
Subroutine LoginError
        {
        /*Function that returns error message if login failed and returns user to login function*/
        }
Subroutine SearchFlightDatabaseCity(Parameter CityA, CityB)
        {
        /*Function that searches flights for ones that depart CityA and arrive at CityB*/
        }
Subroutine SearchFlightTime(Parameter FlightList, Parameter Time)
        {
        /*Function that searches the list of flights for the one that is closest to the time
        parameter*/
        }
Subroutine FlightsError
        {
        /*A display function that communicates that no flights were found*/
        }
Subroutine DisplayFlights
        {
        /*Function that displays flights, when found*/
        }
        /*End Program*/
        }
```

In the type of code that Dijkstra was trying to eradicate, the program would sequentially step through each function, without defining them separately, so the code would look more like this:

```
Program FlightSearchNotStepwise
{
/*Start*/
/*Define variables etc.*/
/*login*/
```

```
Run Database Language Code

Connected = connect("[servername]", "[username]", "[password]") or die("Cannot connect to DB!");

/*Search, where the search may contain many lines of code to get it to appropriately apply the
parameters.  This is also where the functions that display the information would be, using go to
statements to run the display “function” when necessary.*/

If (!empty(connected)

        Do

        GetParametersFromUser(CityA, CityB)

        Assign CityA = “CityA”;

        Assign CityB = “CityB”;

        Run Database Language Code

        Flight List = Select * from FlightSearchDatabase where CityA = CityArrive and CityB =
        City Depart

        If (empty(FlightList))

        Do

                Print(“No Flights match your search Would you like to try again?”)

        If exists(FlightListFinal)

                GoTo line Y

        Else if (!empty(FlightList))

                Do

                        GetTimeParametersFromUser(TimeA)

                        Run Database Language Code

                CountFlightsA = Count(FlightList)

                CountFlightsB = 0

                While CountflightsA > CountflightsB;

                        FlightListFinal = Select * from FlightSearchDatabase where
                        TimeArrival is between Time A and Time B

                Go To line X;

        If !empty(FlightListFinal)

                Do

                        Print FlightList

        /*End*/

}
```

Appendix Three
Explanation of symbols

Throughout this dissertation there are brief examples written in pseudo code. Pseudo code is a compact, informal description of an algorithm, which uses the convention of a programming language, but is intended to be read by people, not machines. Most of the syntax is self-evident, but in some cases the symbolism is obscure. To clarify the examples, I have included a brief description of the symbols used in the dissertation.

Symbol	Description
()	Argument inputs for subroutines or functions in computer science are enclosed within parenthesis.
{}	Curly brackets in pseudo-code delineate the start and end of a sub-routine or function.
[]	I have used square brackets to denote when a number of statements have been left out for simplicity's sake.
/*text*/	Text enclosed by this symbol are programmer comments.
==	The double equal sign is a symbol meaning if and only if.
;	A semicolon signifies the end of a statement.
x = y	This is an assignment statement.

For more information regarding pseudo code, the computer science department at Cornell University has an explanatory handout they use in their courses:
http://www.cs.cornell.edu/Courses/cs482/2003su/handouts/pseudocode.pdf

To see more syntax examples, Wikipedia has a number of good examples buried in an article titled "Template Method Pattern". The following link contains numerous examples of the pseudo code syntax I am personally most familiar with:

http://en.wikipedia.org/wiki/Template_method_pattern