Patterns of Creativity and Composition in Software Development, Music, and Film

George Kelly Flanagin
Bright-Crayon, LLC
PO Box 70966
Richmond, VA, 23255 USA
+1 804 307 2729
george@bright-crayon.com

Arturo Holloway
Bright-Crayon, LLC
PO Box 70966
Richmond, VA, 23255 USA
+1 804 307 2729
arturo@bright-crayon.com

ABSTRACT
This paper explores the relationship between activities typically considered creative, and the role of creativity in software development. By treating software as a composition, and considering that software developers may be composers, we revisit the age-old debate between art and science with a new focus.

The recommendation of the paper is that creativity is a social phenomenon not practiced in isolation. For software developers to be maximally successful, we must let people work cross functionally rather than confine them to silos.

Categories and Subject Descriptors
[**Software Process and Workflow**]: Human activities and processes as they relate to software development.

General Terms
Management, Design, Human Factors, Theory.

Keywords
Creativity, Composition, Music, Film, Art, Science.

1. Introduction
Whatever the production of software "is," we are quite certain it has not been around for very long, at least in comparison to the construction of aqueducts or the playing of the lute. Over the last three decades we have seen a not so subtle struggle to classify it as either an art or an engineering discipline, and there has been no clear winner.

Consider the titles of two early works in the field: The Art of Computer Programming, by Donald Knuth, and The Mythical Man Month, Essays in Software Engineering, by Fred Brooks, Jr. The titles are even more curious once one begins to inspect the contents of each book. The former, with the word "art" in its title, looks like a book written both by and for readers with a doctorate in mathematics. The latter book, with the word "engineering" in its title, is conversational in tone, pleasantly slim, and can easily be read by readers who never plan to write a single line of code.

The management ranks of software development usually share the tendency to classify "programming" with the engineering disciplines. In 1992 a senior Hewlett Packard manager told the author that "calling it 'art' was the worst thing that could have happened to software. I wish my employees had never seen that title," he said, referring to Knuth's book. Whatever software studies are called in a university, they are invariably allocated space in the engineering or math buildings, and students are usually required to have a substantial amount of their undergraduate work allocated to the same electives that support degrees in mechanical or electrical engineering.

On the art side, we find post-university practitioners of software studies are attracted to words like "guru," and "poo-bah," terms that carry, on the one hand, a religious significance and on the other, a reference to *The Flintstones* or Gilbert and Sullivan's *Mikado*. Self-described "architects" and "senior web designers" also abound.

How shall we usefully reconcile these disparate points of view?

It is the thesis of this essay that this distinction is irrelevant and it is imperative that we refrain from indulging in the politics of defining what programmers do as either art or science and focus on the "compositional" and "creative" aspects inherent to the process if we are to truly understand the nature of software development and improve the curriculum of software studies.

In fact, art and science are much the same.

2. What is composition?

2.1 A working definition of composition

For the purposes of this essay, we will be using the terms "programming," "software design," "software architectural practices," and "systems analysis" with a degree of substitution that may make some readers uncomfortable. There is no harm intended; in fact, it is our additional thesis that these apparently distinct activities constitute something of a gestalt, and that the hope of the entire software discipline rests on our ability as an "industry" to un-stovepipe, de-silo, and generally unify software practice.

It is worth asking, before thinking through this problem, whether or not we have working definitions of "composition" and "compositional activity?" Definitions are always tricky. For the purposes of this essay at least, let us consider a compositional activity to be one in which the interrelation of the parts is more essential than the "material" from which any individual part is made.

A real world example would help!

It is said the essence of a bicycle is in its wheels. In terms of strength to weight ratios, and the sheer difference in mass between the wheel and its load, there is little to compare with the spoked bicycle wheel. A cyclist moving along at race speeds is being supported by little more than 24 to 36 wires in each wheel, each only the diameter of one of the strands in an electrical cord.

The best cyclists prefer to have "hand built" wheels, meaning someone begins with a box of spokes, a hub, and a rim, and strings everything together. Hand built wheels made by someone who is skilled are generally three times as strong as wheels built by robot, even when the materials are identical. Building bicycle wheels is a compositional activity with the wheel being the finished composition. The utility and aesthetic success of the wheel depends not so much on improvements to the materials -- the spokes, hubs, and rims -- as it does on the skill with which these materials are combined.

2.2 Composition of Music.

Most of us are more comfortable using the word "composition" in its connection with music. Considering this to be the case, it makes sense to consider whether there might be a direct and useful analogy between the writing of works of music and the creation of software systems. If true, this would be especially fortunate, considering we can draw on at least four centuries of experience with the former as opposed to a mere four decades with the latter.

It is our contention that there is great value in the comparison of music and software and that in doing so we will gain a completely new appreciation for the struggles of software engineering practitioners.

2.2.1 Three people we can't study so easily: Beethoven, Mozart and Schubert.

The fifth symphony of Beethoven is perhaps the best known musical work in the Western world. It is studied in every music appreciation class from elementary school to *Classical Music for Dummies*. Of it, Leonard Bernstein said its magical appeal lies in the finished work "seeming to pour from Beethoven in a single breath." It bears no trace of the heroic struggle to create it; a struggle that took eight years.

Our desire to discover any existing analogy is complicated by a tendency for the effort invested in finished compositions to be invisible to the casual observer. In both music and software, we spend much of our time polishing the chrome on the final product for aesthetic satisfaction and because of practical considerations. Software goes through an architectural review, and much debugging. The failure to "adequately document" software systems for maintenance and enhancements is no different in its origin than the failure of Beethoven to provide us with ten alternative manuscripts showing his cross outs, erasures, mistakes, blood, sweat, and tears, and other rework.

Many other composers were no better at leaving us a trail of crumbs to understand the compositional process. Mozart appears to have gotten great many things worked out in his head, before jotting them down quickly in comparison to whatever time his musical ideas may have spent in the privacy of his mind. [1]

In his mere thirty-one years, Franz Schubert left behind a pile of unfinished and abandoned manuscripts that equals the completed output of many composers: six unfinished symphonies, several tens of partially completed songs, four operas (none of which are really completed) and many pages of abandoned piano music. Yet very few of these offer any insight into his compositional processes. The implausibility of his writing many revisions is accentuated by the fact that in 1815 alone he wrote nearly 150 completed songs for voice and piano - about one every two days.

2.2.2 Three alternatives: Bruckner, Liszt, and Dvořák.

Considering the forces of erosion have removed many of the insights we might get from three of the best known composers of the classical age, we must ask from whom in the pantheon of great composers *may* we get some enlightenment? We offer three examples to consider, each from approximately the same period in time and the same part of the world, yet each giving us a unique insight on the compositional process as it relates to software design and development:

- Anton Bruckner, 1824-1896, Austrian symphonist and choir master.
- Franz Liszt, 1811-1886, Hungarian pianist, composer and arranger.
- Antonín Dvořák, 1841-1904, Czech symphonist, teacher, and a writer of music for smaller forces.

2.2.2.1 Anton Bruckner

Without giving more biography than allowed by the limitations of space or audience interest, let us say that Bruckner is mainly known for his nine numbered symphonies, and the two unpublished symphonies written earlier in career, oddly referred to as "Number 0," and "Number 00." From our perspective of studying software processes, Bruckner is most interesting because he had a case of "revision-itis" that would make any programmer feel at home.

Let us consider his third (or was it really his fifth?) symphony as a representative example to make our case. It first popped onto the musical firmament in 1873 as a little over an hour of loud and sometimes longwinded horn blowing that made plenty of references to its dedicatee, Richard Wagner. On the last page of the score Bruckner wrote that the work was "completely finished, the night of 31st December 1873." This was to be far from the case. [2]

The Vienna Philharmonic Orchestra (translation: "the users") rejected the work as unplayable the next month. While Mr. Bruckner started "Version 2.0" of this symphony right away in the spring of 1874, he was distracted by other work, and resubmitted a substantially revised work to the same orchestra in 1877. As we would say in the twenty-first century, it bombed at the box office.

Bruckner was very sensitive about criticism of his music, as are many programmers about the quality of their code. It is the consensus of most present day critics and scholars that he revised his works so much that the "final" versions of his various symphonies sound overly much alike. Gustav Mahler begged him to stop with the revisions to the third symphony once Mahler saw the printed score in 1878. [3] Indeed, by the time Bruckner finished or gave up on them, all his symphonies may be said to have been written around the end of his life.

How often do we hear of programmers who work on the same snippet of code for far too long?

Bruckner may have been the first composer to need version control software. Even the year 1877 was not to be the end of the assembly line for Bruckner's third symphony. In 1889 another version (3.0?) was published, now at least ten minutes shorter than the original, and with some portions so drastically altered that they seem completely new rather than derivative. [4] Not too many years ago, musicologists digging through the rubble of Bruckner's manuscripts even discovered another unpublished version of the second movement of the symphony. [5]

Only Bruckner's death stopped the process of *revision*, which, for him, was very nearly the same process as *composition*. His manuscripts are filled with evidence of his struggles in much the same way that programmers sometimes "comment out" lines of code. In both the case of Bruckner's music and software systems, the forensic task of those who have followed is complicated by never being sure exactly what is meant, nor why the changes were made.

In Bruckner, we see the pattern of creativity as well as the potential for its obstruction through revision.

2.2.2.2 Franz Liszt

Liszt's music, like that of Rimsky-Korsakov, could have been said to fall into two categories: the over played, and the unknown. [6] Franz Liszt's motivations for revising his works were most certainly not inspired by any sensitivity to criticism: Liszt's ego and his skill as a performer appear to have known little bound. Along with violinist Nicolò Paganini, Liszt more or less invented the solo concerts / touring performances that can trace a direct path to modern day *Phish* concerts. It is certainly possible that Liszt may have been responding to an internal critic as the source of his revisions to his music, but if that were the case, the cat was never let out the bag.

Liszt's thought process about many of his revisions appears to have been closer to what we now think of as "usability concerns." It is important to remember that Liszt had enormous hands, and particularly during his youth, incredible technique. During his middle age he frequently reworked early pieces, which only the younger Liszt could have played, into something more suitable for mere mortals. [7] As Harlan Mills of IBM said about programming, if debugging is harder than writing the code in the first place, and if you are as clever as you can be when you write the code, how can you ever hope to debug it?

Second, during the nineteenth century, the piano was evolving rapidly, and Liszt felt a need to change his piano music in two counterintuitive ways: The range and dynamics of the piano were getting greater because of advances in construction techniques (i.e., the hardware was getting better). But, there was a removal of features left over from the days of the harpsichord. [8] We can think of this latter problem being akin to the disappearance of little tricks and optimizations we used to encounter in code.

Third, Liszt responded to his own compositional motor, and its interaction with the many people around Europe who played his music. We may recognize Liszt as unusually responsive to requests for enhancements and new features. As new musical ideas occurred to Liszt, he incorporated them into already existing compositions, wrote them out, and sent them off to publishers.

In other words, Liszt was simply meeting his customers' requests.

2.2.2.3 Antonín Dvořák

Let us consider only briefly our final candidate for comparison of the software and music compositional processes. Dvořák was, by today's standards, what we might consider a "free spirit." Although he studied music quite seriously and formally, he felt little pressure to conform. Dvořák was also not a very good record keeper. His personal catalogue of his many compositions is missing numbers in the numbering scheme, and worse still for forensic musicologists, contains duplicated numbers.

Throughout his fairly happy life, Dvořák wrote string quartets: compositions for two violins, a viola, and a cello. It is interesting to note that the one we know as his seventh effort is the first to have been published. [9] Frequently, we view historical figures as always having been successful, and in the case of music, we believe every note from their pens was immediately added to the list of performed works. Not so.

Rather than enter the vicious cycle of revisions pioneered by Bruckner, Dvořák decided to try a new attempt at meeting market needs when the publishers rejected his early products. His first attempt at quartet writing dates from 1862, and was modeled somewhat on the quartets of Schubert forty years before. The next three form a kind of set, and are attempts to be "modern," which at the time meant "music like Wagner and Liszt." Publication was not to be for these early works.

Undaunted by failure, he began on another pair which has a bit of the flavor we have come to associate with Dvořák: folk melodies. The one we know as the sixth quartet was abandoned before it was quite finished, and Dvořák began the seventh quartet in 1876, fourteen years of persistence after the first attempt.

From the point of view of learning something about composition in software development, it is important to see Dvořák had multiple teachers and models. He learned from each, and assumed his difficulties in getting published were not due to any inherent problem with his ability.

Fortunately, Dvořák was even less concerned about covering his tracks or appearing to have "done it right" from the start. His early quartets survive and can be studied as examples of how he learned to compose. Brahms, on the other hand, said that he burned his first twenty attempts.

3. Composition in Software Development.

Most of the published work on software practices tends to be of the "how to" variety. "How to" books sell well but it frequently appears that the most that can be said about software development is that it can be learned in twenty-one days from a single large book, or that the dunderhead's guide to a particular programming language will really tell the reader how to use it to build commercially viable products. Existing software systems do not support all the claims made for the "how to" model but before we can improve it we need to address the question: *Where should we start?*

3.1 Brooks's advice for improving software development.

Brooks offers in his essay "No Silver Bullet," now reprinted in the silver anniversary edition of *The Mythical Man Month*, that there *are* a few steps one can take to accelerate the progress of "great designers," ones that he says are "an order of magnitude more productive than the acceptable norm." His insights are powerful and controversial, and it is worth it to see how they stack up against a description of software development as a compositional practice.

Brooks's advice has proven to be rather controversial over the years – this particular bit of it first appeared in 1986 – and it is worth it to ask ourselves, "*Why* is it controversial?" One starting point is to see if Brooks's advice is consistent with what we have proposed about the theory of composition as a metaphor for software system design and implementation.

Somewhat abridged for brevity, Brooks's prescriptions are these: [10]

3.1.1 Identify the best people early, regardless of their experience.

Item one is certainly consistent with what we know about the production of great art, or at least great music. There are not too many teenagers who have written novels because telling such a story requires the types of life experiences that cannot be accelerated. However, plenty of teenagers have proven to have a great ability with things that are less mediated by life events: music, math, even poetry, and definitely software.

However, the modern hiring and management practices do not generally spend much time identifying top performers early in their careers. There are several reasons for this lack of attention, and most of the reasons cut both ways: average tenure of employment in the software field is rather short; many companies do not believe they are interested in the top rung of performance, and do not see it as their jobs to create an environment in which peaceful working conditions can be maintained between the rank and file and a non-management elite.

3.1.2 Apprentice them to the best people you have.

Item two has, for at least a couple of thousand years, been the way real knowledge and craft were transferred from master to student. It is not, however, the way things are done in most employers' workplaces. One must keep in mind the silver bullet being sought by employers is not one that would magnify the differences between software developers. Not only are there political arguments about whether one must treat all employees equally, but there are economic arguments for generic training for the masses at reduced cost per person.

3.1.3 Let them try their hands at a number of elements of the software development world.

Suggestion number three is in serious jeopardy at even medium size employers, as more companies move toward rigidly defined job descriptions that attempt to make specialists from recent college graduates as soon as the hiring process is concluded. However, suggestion three is alive and well at smaller places of business out of necessity; in fact, the need and desire to cross between different niches of software development is often what defines the de facto "small company" mentality.

A senior project manager at a large company related the following story:

> "I was assigned a small project that really needed doing; which is why they gave it to me. The person requesting my services told me that he had also assigned an analyst to the task for 40 hours. Thinking of the need to begin work as soon as possible, I said, 'Look, if it is only 40 hours, I can just do the analysis myself.' I was told that doing the analysis wouldn't be a good use of my time although I have actually published an article on the critical subject matter of the project."

In the case of this organization, the need to maintain the division of labor clearly exceeded the need to have the best person work on it.

When we look back to our examples in music, we find the very best composers wrote music for a variety of instruments and voices, even when they had a clearly defined "favorite axe." In fact, what can be said to make Beethoven, Mozart, Schubert, and J.S. Bach more popular than many other composers is that they were proficient in a variety of musical settings. It is thus easier for the casual listener to find a piece by one of them in an idiom that listener is comfortable with.

Does this boundary crossing exist in software development? Not often. An astute listener can pick up a clue from the vocabulary: Particularly during the last couple of years, the word "resource" has become a euphemism for "person," as if to imply that anyone could be assigned the task. Not too long ago resources and staff were separate items. Combining the advice to allow promising newcomers broad exposure along with the suggestion that these promising newcomers be identified is asking for a social revolution that most managers do not want to address.

3.1.4 Provide opportunities for the best people to work together.

The core of the social revolution is contained in suggestion number four: let the brightest and best people work together. The self-selection phenomenon of intelligence is thoroughly documented in the book *The Bell Curve*, by Herrnstein and Murray [11]. *The Bell Curve* is a subject of considerable controversy, and even accusations of racism, but the ability of intelligent people to find each other is without serious doubt. For example, a few years ago the following snippet was overheard at a café on University Drive in Palo Alto, California: "The *good* people go to Stanford, get their BSCS, and find jobs as top engineers at Hewlett-Packard. The *really* good people go to Stanford, drop out after a couple of years, and start a company."

There are certainly top designers at many large companies, not all of which are companies directly involved with the production of software as their core business. Many extraordinary people choose these companies for the same reasons as their less talented counterparts: steady income, social environment, location, and access to medical benefits (for companies in the United States). For these people, industry conferences provide a frequent source of the type of intellectual interaction Brooks tells us sharpens one's creativity.

Very little great work is done in settings like the isolation of Walden Pond, or in "work at home" situations. In fact, Schubert set Goethe's poems to music; Mendelssohn revived interest in Bach's music, and ate dinner regularly with Robert Schumann, and Wagner married one of Franz Liszt's illegitimate daughters.

It is sad many people go through their lives believing the great composers lived and still live in some kind of vacuum apart from other artists. Good people will get together; the question for management is what its role will be in providing the environment and atmosphere to facilitate this endeavor.

3.2 Is a software system a composition?

There are two questions left to consider in this essay. The first is: *If we accept that software development is a compositional process, filled with revisions, struggle and brilliant insight followed by explosively rapid work, do the resultant software systems fit our criterion for being a "composition?"*

We think the answer is "yes." First, consider that the degree to which software systems please the users is not tightly bound to the materials used to construct them. Seldom has any customer or commissioner come forward with a request that says "I want a software system that is object oriented," or "I want a software system that is built with Enterprise Java Beans." Instead, they tend to make requests to have software systems built that solve some particular problem: "I want a system that will track my packages," or "I want a system that prints many address labels on a single sheet of paper."

Second, consider the fact that the current quest for formal processes and methods has continued along its own line, regardless of changes in the underlying technology. Managers, theoreticians, and tool smiths are still engaged in the same quest that they began some years ago.

The numerous failures of the current fantasy of achieving good results by combining (1) a generic project manager and (2) a little drive-by architecture with (3) a cookie cutter methodology, is not too far from a tacit admission that the software system is a composition. Unfortunately, our industry is not exploring the compositional process, nor are we working on improving any methods by which it is transferred from one practitioner to another.

Which brings up the final question: *What do we know about the way software developers actually work?* There are still very few books on this subject, although there are more books than can be read prescribing how software developers *ought* to work! One book that is rather old but still useful is Susan Lammers's book *Programmers at Work*. [12]

Although it is not essential to cite every single reference in Lammers's book, we notice a few patterns in the responses of the industry leaders she interviewed. For example, most of the people interviewed report that a picture is indeed worth a thousand words, at least in the sense that the use of pictures is an approach to solving problems from a high level of abstraction.

All of the people interviewed tend to view the programming languages, operating systems, and hardware as a medium in which they seek to express themselves. Even though they see the medium as subject to change and "improvement", this fact doesn't really change their opinion of what they do or how they will succeed.

All who speak on the subject side with Brooks in reporting that conceptual integrity is a key part of the system's success, and we see that this is already known to be a fact in the composition of music. None of the people Lammers interviewed seem particularly repulsed by the idea of coding, testing, or scheduling – activities that we currently find in silos of isolation. And in music, we see that composers generally see themselves as responsible for their work from conception to the published work. Finally, composers and software developers all feel that communication and interaction are the keys to the creation of satisfying products.

3.3 Creativity

It has been said that a software developer can be identified as a person who repeats an experiment while expecting a different result. An old adage of software testing says, "If it only happened once, it didn't happen," a kind of cruel compliment *and* complement to the first barb. Through our occasional insistence to look at software development as an activity to be either engineered, or abandoned as inscrutable art, *our industry is repeating the same experiments in learning*.

We claim the software process and the resulting software are not much affected by the programming languages or the formal methods. If this statement is true, it is important to entertain the idea that the failure of languages or methods to make tremendous leaps is because they are merely inappropriately suited to the problem we happen to be solving at the time.

In other words, they are not evil – just misguided.

3.3.1 Reliance on creativity in industries other than software.

Plenty of businesses rely on creativity, but still make deadlines. Engineers are constantly coming up with creative solutions that either fix problems or provide us with new products. Advertising agencies prosper by providing creative content on tight schedules enforced by publication deadlines. Most of the jobs in music are on the commercial side, where jingles must be exactly 28 seconds long, or the music for a movie must exactly fit the film that has already been shot.

In fact, let us consider two relevant and relatively recent examples from the film industry, *Jurassic Park* and *The Matrix*.

The film industry is a good one to consider because we can view the role of the director as involving a good bit of composition, and also because the director's role is generally viewed as central. In addition, the activities of software development and filmmaking are further intertwined by the film industry's tremendous reliance upon the capability of software systems to generate and edit the images we see.

3.3.2 Creativity in Jurassic Park

In the short film *The Making of Jurassic Park*, [13] there is an excellent discussion of the creative challenges shared by the traditional animators and the early proponents of computer generated graphics. Stephen Spielberg started out to film his movie using the traditionally constructed physical models; toy dinosaurs, if you will. When Spielberg saw the capabilities of the computer models, he decided to change technologies, leaving the traditional animators with the problem of adapting to a completely new technology. *Remember in this analogy that "animation" is the real skill, one that is more central than the techniques used to realize it.*

The animators report on several directions chosen to cope with the new technology. In one sequence, we see the animation staff filming themselves running around the parking lot while holding their arms out front like the dinosaurs they wanted to model. Shortly after we see one of the animators running a computer simulation in reverse, telling the viewers that he could see several problems with the dinosaur's gate when he watched it backwards.

For human factors engineers, the most interesting moment may be the point where one of the animators shows a "dinosaur input device." The animators were not familiar with the console interface to the software, and asked the computer graphics staff to create a small abstract model of a dinosaur. By manipulating the model in a manner with which they were accustomed, the animators were sending information to the software about how to move the cyber-saurs.

Creativity need not adversely impact productivity; it just might spark it.

Let's take a quick look at the success factors. (1) The technical staff and the users were willing to experiment and be creative, even in a situation where the success of the movie was at stake. (2) The animators sought compositional advice about dinosaur steps through an examination of their own walking, something they could readily observe. (3) The compositional aspects of the film itself were furthered by the newly discovered "ease" of working with a novel user input device that was actually a "throwback" to the older way of doing things.

3.3.3 Creativity in The Matrix

Since the introduction of the DVD, we have learned a great deal about the making of many films. The DVD of the 1999 movie, *The Matrix*, also includes a short film about its making. [14] *The Matrix*, like *Jurassic Park* won several awards for its innovations in technology, and they are covered not only in short film on the DVD, but also in detail by the directors on an alternate audio track that lasts the length of the movie.

In perhaps the most interesting example of the compositional process at work, *The Matrix,* a movie using "technology" to make real actors appear to be comic book characters, used technology similar to that used by *Jurassic Park* (a movie preceding *The Matrix* by six years) to make characterizations of non-existent creatures appear to be real.

For example, there was a desire to show "bullet time," a device to, among other things, allow various characters to appear to leap into the air and remain briefly suspended while the rest of the universe proceeds at the usual pace. In the accompanying short film we learn how ordinary cameras were arranged in a semicircle to take one frame each. These hundred or so images were later added to the film to stretch the passage of time, allowing Carrie-Anne Moss to *float like a butterfly and sting like a bee*, if only for a second or two.

Again we see the nature of composition manifested in the directors being more concerned about the aesthetic affects on the viewer than the technology used to create it. Again we see creativity functioning in a controlled environment of budget and schedule. Again we see something working in full, appearing to be a seamless whole when actually it is assembled from countless parts.

Can composition and creativity be taught and learned? The answer is certainly "yes."

4. Conclusion

We must look away from the debate pitting art against science. We must place our focus on the compositional and creative aspects of building software systems. We must derive examples suitable for how we teach abstract mirroring of problems and solutions from the real world. By doing these things we will get much farther than if we continue to look only for improvements in materials.

Any depth of knowledge in the use of a technology will sooner or later become obsolete, no matter how popular the technology may be today. In fact, the accelerated rate of change in the media that support software makes it all the more important that we do this.

In the past, and in other professions, the media might well have a lifetime that exceeded the lifetime of any individual craftsman. Currently we have a situation in which one can just barely become an expert with a language as complex as C++ before it is replaced with something else, like Java. We can no longer rely on the constancy of our technologies.

By creating silos in the workplace, we have abandoned the learning of integration and composition. It is rare to find anyone who has the "big picture." This strategy of divide and be conquered is contributing to the cost overruns and lowered quality that we have today. Careful mentoring in the compositional and creative processes, combined with an acceptance of the transitory nature of languages, operating systems, and hardware, will create more adept software composers. These composers will then provide better, more satisfying software systems.

5. REFERENCES

[1] In a letter Mozart relates "I do not hear in my imagination the parts successively, but I hear them, as it were, all at once. … For this reason the committing to paper is done quickly enough, for everything is, as I said before, already finished; it rarely differs on paper from what it was in my imagination." From The Letters of Mozart, edited by Hans Mersmann, Dover Publications, 1972, p. vii.

[2] Eliahu Inbal, liner notes to Bruckner Symphony No. 3, Teldec 8.42922.

[3] John Warrack, liner notes to Bruckner Symphony No. 3, Deutsche Grammophon 431 684-2.

[4] Richard Osborne, liner notes to Bruckner Symphony No. 3, Deutsche Grammophon 413 362-2.

[5] Georg Tintner, liner notes to Bruckner Symphony No. 1, Naxos 8.554430.

[6] Richard Taruskin, "The First Russian Symphonist?", included in the liner notes to "Rimsky-Korsakov, 3 Symphonies," Deutsche Grammophon, 423 604-2, 1988.

[7] Leslie Howard, "The Schubert Transcriptions, Vol. III, The Complete Music for Solo Piano, Volume 33," Hyperion CDA 66957/9, 1994.

[8] Leslie Howard, "Douze Grandes Études, The Complete Music for Solo Piano, Volume 34," Hyperion, CDA 66973, 1994.

[9] Prof. Dr. Jarmil Burghauser, "Dvořák and the String Quartet," Deutsche Grammophon, 1977.

[10] Frederick P. Brooks, Jr., *The Mythical Man Month*, Addison-Wesley, 1995. p 202.

[11] Richard Herrnstein and Charles Murray, *The Bell Curve; Intelligence and Class Structure in American Life*, Simon and Schuster, 1994.

[12] Susan Lammers, *Programmers at Work*, Microsoft Press, 1986.

[13] Jurassic Park, Collector's Edition, Universal Pictures, DVD, 2000.

[14] The Matrix, Warner Brothers, DVD, 1999.