The Artist in the Computer Scientist

More Humanity to our Research

Daniel Cukier
IME-USP
danicuki@ime.usp.br

Joseph Yoder
The Refactory, Inc
joe@refactory.com

Abstract

Art and Science are usually seen as quite distinct tasks and not supportive of each other or similar at all. Isn't art all about creativity and abstract beauty, while computer science is about logic, truths and problem solving? Can these two practices really be related in any way? Our primary objective is to show the benefits of arts to software development. First we reflect on the concept of how Art and Science are similar. Then we report our thoughts about the relation of different types of art to Computer Science such as theater, music, painting, and poetry.

Categories and Subject Descriptors A.0 [General]; D.m [Miscellaneous]: Software psychology.

General Terms *Design, Human Factors.*

Keywords Art and Science, Computer Science, Theater, Drawing, Painting, Poetry, Art.

1. Introduction

Art and Science are not commonly seen as similar activities or related practices. In fact, it is more common to find works that relate these themes as being disconnected. However, even in the beginning of the *Communications of the ACM*, in 1959, the editors highlighted the following objective for the ACM periodical [2]:

> *If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected.*

Donald Knuth, who wrote one of the most famous Computer Science books, *The Art of Computer Programming* [23], affirmed that computer programming left the field of Arts to become Science for just one simple reason: we started to call it "Computer Science" [24].

The word **Art** comes from Latin *ars*, which means "skill." The Greek term would be *τεχνη*, root of the words "technology" and "technique." There was a time when humanity did not differentiate Arts from Science. People in medieval times were responsible for teaching the so-called "liberal arts," which were the seven liberal sciences: grammar, rhetoric, logic, arithmetic, geometry, music, and astronomy. From these seven, three are considered very close to Computer Science (logic, arithmetic, and geometry).

Knuth also said:

> *Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it [...] there is a huge gap between what computers can do in the foreseeable future and what ordinary people can do. The mysterious insights that people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an art. [...] The science without the art is likely to be ineffective; the art without the science is certain to be inaccurate. [...] We need to combine scientific and artistic values if we are to make real progress* [24].

In literature there are many situations that mix (and sometimes confuse) the words "Art" and "Science." For example, there are the books *The Art of Piano Playing* [28] and *The Art of Playing the Piano Forte* [20], but we can also find *The Science of Pianoforte Technique* [12], or even *The Art of Piano Playing: A Scientific Approach* [25]. We can also cite *The Science of Being and Art of Living* [38].

In "A Short Introduction to *The Art of Programming* [9]," Dijkstra says that programming requires good taste

Onward! 2011, October 22–27, 2011, Portland, Oregon, USA. Copyright © 2011 is held by the author(s). ACM 978-1-4503-0941-7/11/10…$10.00.

and style and that a professor cannot teach a student to write specific programs, but has to help his pupils find their own styles.

In *The Mythical Man-Month: Essays on Software Engineering* [4], Fred Brooks compares the programmers' way of thinking with the poets'. He affirms that both types of work are slightly removed from pure thought-stuff. Richard P. Gabriel says that every programmer should be trained as an artist [19]. Paul Graham cites many similarities between programming and painting [18]. If Computer Science is Science and Art at the same time, we should include more Art in the daily life of computer science students, as is suggested by Prof. Valdemar Setzer in *Um Antídoto Contra o Pensamento Computacional* [36]. The famous astronaut Mae Jemison agrees that we should definitely join Sciences with Arts [22].

We aim to show the benefits of Arts as applied to Software Development. To do this we reflect on the relation of different types of art with Computer Science. The types of art we will examine to illustrate our point are theater, music, painting, and poetry.

2. Theater

One of the most complete forms of artistic expression is theater. This is because quite often theater joins many forms of art. Theater brings together elements that enable the actor to get in contact with poetry, music, and dance.

Not so long ago it would have been very difficult to imagine how Theater and Computer Science could go together. Nowadays it is easy to see that Computers, and in a wider view, technology can contribute to theater [33]. All theatrical production requires some technological resources, such as lights, sound, scenery, etc. Any theater play will use (directly or indirectly) audio software, and design software.

The objective of this work is to reflect about the other side of the coin: how the Arts can contribute to development of these technologies. Art creates technology, because Art comes from creative human perception, something completely disconnected from technological tools. Human beings are born with creative spirit. Donald Winnicott says [37]:

> *It is through creative perception, more than everything, that an individual feels the dignity to live his life.*

What most arts propose (and what theater is in our vision) is the creation of a ***ludic*** (or playful) space and social interactions. This open space, when first created, will be used to receive people's ideas made material. Winnicott developed a theory in which he shows that during play, and maybe only during play, people experience the real freedom of creation. Whatever kind of creation it may be (technological, in our case), it will be probably more authentic if it arises from within a ludic space.

In 2000 a one semester course of dramatic lectures was created as part of the curriculum of the Institute of Mathematics and Statistics at University of Sao Paulo [35]. Since 2003 this course has been an official optional course for Computer Science students. Since then, more than 400 students have successfully taken it. During that period, students also put on five theater plays with the help of Professor Jolanda Gentilezza.

Computer Science students are required to deal every day with so-called unquestionable truths, quite often mathematically proven. This overdose of logic and digital thinking contributes to keep hackers far way from human feelings and personal relationships. Furthermore, the nature of software development does not follow a linear, well-defined process, but goes through cycles of improvements that aim to fulfill clients' needs. On the other hand, Theater has the role of balancing logical thinking with the more unpredictable and non-logical human side.

The creation of a good piece of software cannot be streamlined as seen in manufacturing processes. If this *were* the case we would have seen software factories emerge during the 90's. Instead, software is, in general, more of a creative developmental process rather than a manufacturing process. Often, experimentation is done to try different approaches—specifically when the goal is to create something new and unknown. A spike solution is done to figure out an answer to a tough technical or design problem. Excellence is achieved after many such tries and variations of the same theme, in a search for perfection.

At Disney there are hundreds of actors whose job is just to make people have unforgettable and wonderful moments and memories. The requirements of what is a "wonderful moment" change depending on each visitor. The actors job is to (1) discover the high quality experience that each visitor wants to have and (2) make sure that the visitor really has this experience [34].

In the software area, there is no unique solution or technology that solves all problems. There is no silver bullet [3]. It is fundamental for programmers, besides having technical knowledge, to be creative and to know which technology is most appropriate for each particular situation.

The quality of service vision (and today software is also a provided as a service [32]) considers that each customer has a different idea of what a quality experience is. The difference between providing a service and creating a product is that, in services, meeting each customer's needs is an unspoken requirement, while in production lines, such variation is the enemy.

In abstract terms, the nature of software development is related to creating personalized solutions and establishing an iterative improvement process. Theater incorporates ludic elements that facilitate developing human skills—

indeed elements that facilitate creation of adaptive solutions for each audience. Theater also has in its nature an iterative improvement process: each rehearsal aims to refine more and more the character, the text, and the scenes. The evolutionary approach is similar for both theater and software development.

Another benefit to computer science education of theatrical practice is the improvement of communication skills. Contact with high quality texts fosters linguistic development. Acting contributes to developing communicative leaders. One of the main factors for success in companies is having high quality communication between its employees.

Theater also teaches how to deal with emotions. It helps with the hard task of recognizing feelings and realizing their consequences for both mind and body. Besides recognizing feelings, each actor develops empathy, the capacity to recognize other people's feelings. This helps teach how to engage in teamwork and how to build collaborative spatial environments.

By experiencing theatrical activities, people have the opportunity to develop their creative skills. In theatre, creation without limits is permitted—even encouraged. There, you may invent and be different. Every good programmer wants deep down inside to create good things—to create good software. And increasing creative skill assists this process for all disciplines, including computer science.

Logic, Mathematics, Computer Science—they are tools, techniques. Knowledge of technique is fundamental. But there is a knowledge that comes before technique and that needs to be developed: the knowledge of what is not defined (or is badly defined); contact with uncertainty, non-logic; living the objective sensation experience of one's own body; and interactions with others. These are experiences that develop creativity and communication skills. By unifying creation and communication with technical knowledge in a company's workforce, we have the ideal environment for great technological products.

Markets are extremely competitive. Few people are able to, by themselves, develop great software. Modern day systems are usually large and complex, full of features, and supposed to be easy to use. Furthermore, it is becoming more important for systems to interact with each other. Many teams and many people are responsible for developing and maintaining these complex systems. These groups of people have a variety of relationships [11]. In the future, things will tend to become even more complex, possibly with trillion-line code systems running simultaneously in millions of computers, interacting with each other in realtime [29]. No human being is able to develop by themselves these kinds of systems [15]. In order to deal with this complexity, people will need to communicate more efficiently, sometimes having to speak using a Ubiquitous Language [11]. That is why it is so important for individuals to develop their own skills in communication. Theater is just one discipline that helps people to develop communication / community skills.

After a retrospective of the first dramatic lectures class in IME-USP, the teacher, Jolanda Gentileza, came to the following conclusions after chatting with students:

- The most pleasant surprise was that all the students reported a decrease in their shyness.
- All of them liked bodywork, reporting it as a "truly relaxing experience, great for the relief of weekly tension"—giving them the opportunity to realize what Jolanda calls "shyness musculature."
- They thought that lectures were "a lot of fun" and they would like to have more lectures of that kind. (They read "Lysistrata" by Aristophanes; "King Oedipus" by Sophocles; "The Imaginary Invalid" by Molière; and many other poems and small pieces by many other authors).
- They liked the "theatrical games" very much because (they used this phrase) "to be in shame together" makes you lose your shyness.
- They asked for less theory, but this was obvious—students will always ask for less theory. Anyway, it is always possible to hide theory in practice. So they were right, because they passed their whole day in theories.

3. Music

Analyzing the many relations between Music and Computers (and Mathematics) would require its own separate article. There are dozens of books about Music and Computers. We limit our scope to some points that we consider important to make our point.

In previous research [8] we verified with both dramatic lecture students and IT Industry professionals that music is the most desirable and common form of artistic expression enjoyed by software developers. While many software developers reject many other kinds of art such as theater or dance, music is almost always unanimously appreciated. For this reason, we can easily use music to communicate ideas. Music can be an excellent communication media. As described in the pattern Do Art [8], we can compose a song that contains, in its lyrics, some elements representing whatever idea we want to express.

Similar musical tastes join people together. By singing (or by playing an instrument), we can get in touch with our own ludic and creative sides. Environments that have a musical background tend to be more informal.

Sometimes, composing a song is a standalone process, where the composer works alone—at first. However, composers rarely finish their compositions alone. There are many additions that must come after the music is composed. Sometimes the composer needs to pair with some-

one else to create the lyrics. Then comes the arrangement, the harmony, the choice of instruments, the recording of performances (together or on separate tracks), the mixing, the mastering, etc. There is a long road from the time music "appears" in a composer's mind to the time it is ready to be listened to by an audience. In fact, the process of first creating a song through putting the final piece into production requires many people collaborating in order to be successful. Of course the song writer(s) almost always own the core of the material and gets most of the credit.

Music composition is also an exercise of many trials with failures. We try a chord and realize that it does not fit well in that part. Then we "refactor" it to another chord. Then the melody needs to have one note changed to fit the new chord. From the first draft to the last version there are many intermediate solutions. The composer usually shows the song to friends or other composers, requesting feedback or comments (a process like peer reviewing). And just like software or poetry (see Section 5), it is never finished; many composers give up improving it when they think the song is good enough.

In software we find a similar situation where there are many dependencies and collaborations in order to design, implement, and put a system into production. Even if the software has been designed, programmed, tested, and is working in the development environment, we still require DBAs to create the production database, the operations team to provide a production machine, the network and security team to provide the IP addresses and configure the firewall, people to supply and tune the load-balancing infrastructure, etc.

In software there is a creative and interactive process where a team of developers, analysts, testers, architects, project managers, and others collaborate to orchestrate the development and creation of a system that can be ultimately deployed into production. Without good teamwork and collaboration many projects fail. So, like music, putting any significant piece of software into production is much more than just a single person working alone.

4. Drawing and Painting

A core practice of any professional software developer is to regularly practice design. Let's take a closer look at what design means, and then understand the relations between drawing, painting, and programming. Richard P. Gabriel, brings some interesting definitions of *design* [15]:

- A mental plan. A plan or scheme conceived in mind and intended for subsequent execution.
- Design is a plan for how to build a thing. To design is to build a thing in one's mind but not yet in the real world—or, better yet, to plan how the real thing can be built.
- Design is the thinking one does before building.
- Designs are instructions based on knowledge that turn resources into things that people use and value.

The above definitions can be seen as common tasks done by a professional programmer. When a programmer comes across a problem to be solved, she mentally plans a possible solution for the problem. After this planning, the programmer transcribes the "mental plan" into an algorithm using her favorite programming language (or sometimes the language imposed by the context).

We can define the process as follows: (1) the arising of a new problem, (2) mental design to solve it, and (3) solution execution. Then we ask: is this solution by this programmer the best solution for this problem? Is this a definitive solution, or can it deteriorate over time? Here we can identify a limitation of this process. The limit is on the human being's mental capacity to think logically about the problem and propose a logical solution for it. What if, after some time, this programmer realizes that someone else had already passed through this problem space and that the other person's solution is better? It would be a waste of time to try to (re-)solve the problem. Unfortunately many developers have the "not invented here" disease. A programmer to code can be like a dog is to a fire hydrant.

Programming and painting have many similarities [18]. First of all, both painters and hackers are creators. Like composers, architects, and writers, painters and hackers try to create good things. Hackers want to program, creating well-made and usable software. Sometimes the academy blocks this hunger for creation—because sometimes the parameter used to evaluate people is the number of articles or publications. A good piece of software is not always an appropriate subject for an article or technical paper. Besides this, the subject is not always original. An undergraduate student in art does not need to write an article about the painting just painted. The work speaks for itself. Why can't Computer Science students be evaluated by the quantity and quality of software produced? Perhaps for the same reason that the aptitudes of a pupil in school are measured by standardized, multiple-choice tests while the productivity of programmers is measured by the number of lines of code produced. These tests are easy to measure.

How can we truly measure the work of a hacker? Sometimes it is the same as evaluating the value of a painting: it needs a good sense of design to be judged good design. This is the hard part. One possible solution for evaluation is to notice that beautiful things tend to last for a long time, while ugly things tend to be forgotten. The problem is that, sometimes, the time required to accurately judge a work this way is longer than a lifetime. Many great artists do not have their work recognized until long after their deaths.

Talking about theory, hackers need to know computer science theory in the same way painters need to understand the natures of oils, acrylics, and watercolors. The first version of a painter's work is just a draft. The painter looks to the draft and, from that, starts to improve it.

Hackers do the same. They create a small piece of code. They run the code and watch how it works. They make adjustments. Sometimes, a programming language is the concrete (alterable and erasable) medium programmers use to help think about the programs they want to write—instead of as a means to express programs already designed another way (in their heads, perhaps). Programming languages viewed this way would be like a pencil, not a pen.

5. Poetry

"Poetry is at the margins of understanding, in the fractures of our reality, in the space between order and chaos."
Richard P. Gabriel [16]

A computer program can be considered the expression of an abstract idea in a specific (programming) language. To prepare a program can be like composing a song or a poem [24]. The programming language is just the tool to transform a real world idea into something that can be interpreted and processed by computers. If you ask the computer "computer, please, order this list of names alphabetically," probably the computer cannot do it. But maybe someone could write a program in a language understood by the computer and ask it (in this language) to order such list of names.

Suppose a hacker perfectly understands how the **quicksort** algorithm works. The **quicksort** algorithm describes well-defined steps to order items in a list. The algorithm can be abstracted in three steps as follows:

1. Choose any element in the list and call it the *pivot*;
2. Re-arrange the list in a way that all items that come before the pivot have a value less then the pivot, and all items after the pivot have values equal to or greater than the pivot;
3. Repeat the process for each sublist—that is, for the sublist of items less than the pivot and the sublist of items equal or greater.

These steps are real world ideas (*list of names*, *divide the list*, *items greater than the pivot in one side*, *items less than in the other side*, etc.) and were described above in plain English. If we explain (in English) the above steps to a 12-year-old boy, he will probably know how to execute these steps and order a list of names using **quicksort**. If we put these words in a computer, it will not understand and will not do the operation we expect. To "talk" to computers we need to speak its language.

Now suppose we give the following phrase to the young boy:

```
sort [] = []
sort (x:xs) = sort (filter (< x) xs) ++ [x] ++
sort (filter (>= x) xs)
```

The boy would probably not understand a single thing about this phrase. Now, what if we show the same phrase to a developer or hacker who knows the **quicksort** algorithm? Unless the hacker understands the Haskell programming language, he very likely might not understand what this code is all about. A programmer might know that it's about sort because of the names used but still would not know what sorting algorithm it represents. A computer could interpret and understand the program in Haskell if we give it a "Haskell Interpreter"; that means: if we "explain" to the computer how the Haskell language works. Similarly, if we could explain to the young boy how the Haskell language works, he would also (might) be able to understand the code. We are talking about **language learning**. In other words, how can we express ideas by using a language? But, how does one learn a language? By speaking? By listening? By writing? By reading? Most would agree that all these things are necessary in order for someone to learn a language.

Is it possible to express all our ideas in a language when we are very familiar with the language? We know that there are some words that exist only in specific languages and cannot be translated. Every language is limited by its grammatical rules and orthography. Limits arise not only from these rules, but also from its vocabulary. When poets *think* in poetry—(especially) in terms of metrics and rhymes—they limit even more their possibilities of expression. Poets "fight" every day their language when trying to express their ideas. It is a constant fight.

Similarly, programmers "fight" all the time with their programming languages, including their (the programming language's) limitations. Programmers work at making the computer understand messages and commands. A programming language is composed of a few symbols and syntax, and these symbols need to be perfectly arranged according to the language's syntactic rules to represent the specific meaning a programmer is trying to convey. At times, it can seem almost like an outrageous task that requires a lot of creativity and magic from the programmer.

Every Computer Science problem can be solved in infinitely different ways. Hackers can be considered extremely creative people. They are creators. They are inventors. The science of software has not been studied for that long (a little over 50 years). A great amount of current software is completely new. The most interesting software is software that has yet to be created. Hackers just want to create fun new systems. Being a creator, perhaps a programmer should be trained in the same way creative people are

trained, such as artists and poets [19]. How are poets trained? They study famous poets and their work, such as: Fernando Pessoa [31], Hilda Hilzt, Federico Garcia Lorca, Carlos Drummond de Andrade, Lord Byron, Vinicius de Moraes, Pablo Neruda, and Manuel Bandeira. Why do we not do this in software? Shouldn't Computer Science students study the life and works of well-known computer scientists and software developers [5] such as Alan Kay (Smalltalk), Ralph Johnson (Patterns), Donald Knuth (Algorithm analysis), Mark Spencer (Asterisk), Linus Torvalds (Linux), Yukihiro Matsumoto (Ruby), John McCarthy (Lisp), James Gosling (Java), Guy Steele (Scheme), Dave Thomas, and many others?

Most Computer Science programs do not require students to look at lots of good code from important programs. Sure, there are exceptions—the exception rather than the rule—and such exceptions are usually because of individual, enlightened professors rather than being a general part of the curriculum. Rather than look at the architecture or design of good systems, libraries, and frameworks, most Computer Science curriculums consist of studying algorithms, theory, analysis, and languages—book learning, not doing and practice. Of course these things are important but in general, students do not study in depth the things they will be building. Nor do most curriculums show students how to work well and interact in real, team environments.

Good poets improve their writing skills by writing a lot of poems. The same happens to good programmers. They must write a lot of code to become masters. Great poets, before being great, almost always have studied great poems from many poets. They also will have shown their poems to many critics and other poets for feedback. Their poetry is evaluated and revised many times. A writer will become great after many repeated iterations of this process: read, write, revise, read, write, and revise. This is also true if we want to become good in writing software—we need to practice, we need to read a lot of code, have our code evaluated by others, and we must practice revising our code.

It may look strange, but writing software and writing poetry requires the same type of concentration. It is thinking in possibilities, thinking how to do things the best way, how to be elegant, how to simplify, how to efficiently / effectively use the language. If we look to source code done by extremely talented programmers, we will see eloquence and beauty.

A good way to exercise writing is to work in a distributed project—an open-source software project for example—where a group of developers, distributed around the world, work to create software. In this kind of project, programmers are obligated to communicate by writing (emails, forums, and mailing lists). They need to write a lot of documents and make them available to others. This work helps to develop writing communication skills. Furthermore, writing is also a way to bring more discipline to the software development process [17].

Writing software also demands constant interaction with people who (will) use the software. Communication with these people needs to be of the highest quality possible. We have to know how to understand users' desires and wishes.

In agile methods, "user stories" are artifacts [7] that provide brief descriptions of core system features needed by the client. Extraordinary communication skills, as required in theater and poetry, helps eliminate gaps in communication. Writing software is an art [24] and it takes a long time for one to become a virtuoso.

A good programmer should communicate well with people, not just with computers. It is important to not only understand what people want, but also to be able to explain to them the difficulties, the challenges, the technical limitations and situations that a programmer will go through to create the software. Some people still think that programming is a secondary activity in software development. Actually it is a central, difficult, social, and human activity.

Developing software is an art [24]. As in all arts, it cannot be taught or learned as a precise Science, with theorems and formulas. Like a painting (made by a painter) or a building (conceived by an architect), a software product includes the personal touch of its creator [1].

Today, one of the most important difficulties to overcome when developing complex systems is the team's ability to establish high quality communication with the customer interested in the system. Every system is built to solve (or to help solve) a real-world problem, a problem belonging to the customer's specific domain. Therefore, it is important to define a Ubiquitous Language [11], a catalogue of terms common to all people in a project. This job is not easy. To create a common language that precisely defines the application domain is a challenge, as it is a challenge to know how to express ideas clearly using a spoken language.

Poetry exercises enable people to develop skills with words and languages. Working with poetry demands the poet to pay special attention to verbs, substantives, and pronouns—as well as to develop a keen ability to notice. By doing this, one sharpens the relation with his or her own way of expressiveness and improves the ability to notice what is essential for the client and the software being created; moreover such practice improves the capacity to understand others' messages.

Before developing the capacity to model the application domain, a software designer needs to develop knowledge of the language in which the model will be defined and communicated. There are subtle differences between words that seem to mean the same thing. These small differences can be exactly what will differentiate a good model from a poor or ambiguous one. For example, if I decide to model a system using the term "User" instead of "Customer", there

are important differences between the meanings of these two words—a user uses the software whereas the customer pays for the software to be created (and may never actually use that software). Maybe for someone who is not skilful in English these differences may pass unnoticed. Depending on the domain, one term will always be better than another. If a developer does not know such distinctions, the wrong word might be used and thereby seriously compromise the agreement.

It is also possible to use refactoring [13] for writing texts and poems. Writing is a continuous learning process. While someone writes, that person reads, rewrites, and rereads [14]. Pieces of text are deleted, substitutions are made, and fresh text is written. We could say that a written work, like a painting, is never finished. We just stop working on it at the moment it looks good enough or we run out of time.

Working to improve poetry-writing skills is the same as working to improve creation capacity. Today, software can be considered both creational and artistic. Indeed, poetry can help improve programming abilities by expanding creative abilities. A programmer who decides to learn poetry can certainly improve:

1. Their writing skills.
2. Their knowledge about language.
3. How to explore alternative solutions within a space of possibilities.
4. Their abilities of expressing in a language.
5. Their abilities in programming, because programming is merely a way of expressing mental ideas in a defined language.

6. Enunciations

"Technology inspires art, and art challenges the technology"
John Lasseter (Pixar)

We have shown that certain kinds of art and computer science have similar tasks, states of mind, and ways of thinking. Moreover, both art and computer science not only involve a lot of creativity but also usually intense collaboration. We can learn a lot by examining how people become creative in the arts.

Alvin Toffler said [cited in 6] the following:

> *"[...] Society needs all kinds of skill that are not just cognitive; they're emotional, they're affectional. You can't run the society on data and computers alone."*

Computer Scientists and programmers can clearly enhance their skills by becoming better artists.

Peter Brook, a famous English theater and film director argues [30] that:

> *"In our day, the tragedy of art is that it has no science and the tragedy of science is that it has no heart."*

Albert Einstein had the opinion that "The most beautiful thing we can experience is the mysterious. It is the source of all true art and science." [10] He also believed that "After a certain high level of technical skill is achieved, science and art tend to coalesce in aesthetics, plasticity, and form. The greatest scientists are always artists as well."

In *The Art in Computer Programming* [21], Dave Thomas and Andy Hunt show how programming is more than dealing with machines. It is necessary to know human beings and interpret their wishes, going beyond what is asked. We need to take the right questions to clients in order to discover what they really need. Programming is art and requires more than just technical skills.

Pete McBreen, in *Software Craftsmanship—The New Imperative Design* [27], takes software development as the craft that mixes Art, Science, and Engineering with the objective of delivering effective systems. Robert C. Martin (also known by the software community as "Uncle Bob") also affirms in *Clean Code: A Handbook of Agile Software Craftsmanship* [26] that "a programmer who writes clean code is an artist that, from a sequence of transformations, turns a blank screen in an elegantly coded system."

There are many works that show relations between Arts and Computer Science. The union of these apparently distinct disciplines can bring benefits for both Science and Art and, consequently, to humanity.

"When technology and art are together, magical things happen."
Walt Disney

References

[1] A. Avram and F. Marinescu. Domain-Driven Design Quickly. C4Media, 2006.

[2] W. F. Bauer, M. L. Juncosa, and A. J. Perlis. ACM publication policies and plans. Journal of the ACM (JACM), 6(2):121–122, 1959.

[3] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 20(4):10–19, 1987.

[4] F. P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1995.

[5] E. Burns. Secrets of the Rock Star Programmers: Riding the IT Crest. McGraw-Hill/Osborne, 2008.

[6] T. Ciszek. A Framework for the Development of Social Linking Theory. 2005.

[7] M. Cohn. User Stories Applied: For Agile Software Development. Addison-Wesley Professional, 2004.

[8] D. Cukier. Padrões para Introduzir Novas Ideias na Indústria de Software. Master's thesis, IME-USP, Mai 2009.

[9] E. W. Dijkstra. A Short Introduction to The Art of Programming. Technische Hogeschool Eindhoven, 1971.

[10] A. Einstein. The world as I see it. Covici, Friede, 1934.

[11] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004.

[12] T. Fielden. The Science of Pianoforte Technique. Macmillan London,1934.

[13] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[14] R. P. Gabriel. Writer's Workshops and the Work of Making Things. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[15] R. P. Gabriel. Design Beyond Human Abilities. http://dreamsongs.com/Files/DesignBeyondHumanAbilities Simp.pdf, 2006. Accessed in: 20/06/2008.

[16] R. P. Gabriel and R. Goldman. Mob software: The erotic life of code. ACM Conference Object Oriented Programming, Systems, Languages, and Applications—Keynote Speech, 2000.

[17] R. P. Gabriel and R. Goldman. Innovation Happens Elsewhere. Morgan Kaufmann, 2005.

[18] P. Graham. Hackers & Painters: Big Ideas from the Computer Age. O'Reilly Media, Inc., 2004.

[19] J. J. Heiss. The Poetry of Programming—Interview with Richard P. Gabriel http://java.sun.com/features/2002/11/gabriel_qa.html, 2002.

[20] J. Hummel. The Art of Playing the Piano Forte. Royal College of Music, 1992.

[21] A. Hunt and D. Thomas. The Art in Computer Programming. The Pragmatic Programmers, LLC, 2001.

[22] M. Jemison. Mae Jemison on Teaching Arts and Sciences Together. http://www.ted.com/index.php/talks/ Mae Jemison on teaching arts and sciences together.html, 2006.

[23] D. E. Knuth. The Art of Computer Programming. Addison-Wesley, 1997.

[24] D. E. Knuth. Computer programming as an art. Communications of the ACM, 17(12):667–673, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361604.361612.

[25] G. A. Kochevitsky. The Art of Piano Playing: a Scientific Approach. Alfred Publishing, 1967.

[26] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series). Prentice Hall PTR, August 2008. ISBN 0132350882.

[27] P. McBreen. Software Craftsmanship: The New Imperative. Addison- Wesley Professional, 2002.

[28] H. Neuhaus and K. Leibovitch. The Art of Piano Playing. Kahn & Averill London, 1993.

[29] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidth, K. Sullivan, and K. Wallnau. Ultra-large-scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[30] Y. Oida and L. Marshall. The Invisible Actor. Routledge, 1998.

[31] F. Pessoa. Poesia Completa de Alberto Caeiro. Companhia das Letras, SP, Brasil, 2005.

[32] C. Pinhanez. A Services Theory Approach to Online Service Applications. Proc. of SCC'07—2007 IEEE International Conference on Services Computing. May 2007.

[33] C. S. Pinhanez. Computer Theater. In Proc. of the Eighth International Symposium on Electronic Arts (ISEA'97), Chicago, Illinois, September 1997.

[34] M. Poppendieck and T. Poppendieck. Lean Software Development: An Agile Toolkit. Addison Wesley, 2003.

[35] Y. S. Santos. Teatrona Matemática e na História. Jornal da USP, (588):1, April 2002

[36] V. W. Setzer. Um Antídoto Contra o Pensamento Computacional. http://www.ime.usp.br/vwsetzer/antidoto.html, 2006.

[37] D. W. Winnicott. O Brincar & a Realidade. Imago Editora, 1971.

[38] M. M. Yogi. The Science of Being and Art of Living. Allied Publishers, 1963.SP, (588):1, April 2002.