Computer Science: Craft, Science or Engineering?

JIN SUN
Department of Computer Science
University of California at Riverside
Riverside, CA 92507
Email: jsun@cs.ucr.edu
SID: 860-750-875

Mar 2006

Abstract

There has been a lot of philosophical and pragmatic debate on whether Computer Science is a craft, science or an engineering discipline. This article examines both Paul Graham's claim that hackers and painters have a lot in common and Dr. Tolbert's claim that Computer Science is a craft on its way to becoming an engineering discipline. We conclude that both claims have truth in them and argue that Computer Science is becoming a rigorous engineering discipline while still leaving plenty room for artistic creativity that demands great craft skills.

1 Introduction

According to the glossary defined in the letter "High Performance Computing and Communications: Technology for the National Information Infrastructure" prepared by the Committee on Information and Communication (CIC) of the National Science and Technology Council (NSTC) [1], Computer science is the systematic study of computing systems and computation. It includes theories for understanding computing systems and design methodology, computational algorithms, testing of concepts, analysis, implementation and verification. Despite the book definition of Computer Science and other similar definitions, there are in fact a lot of debate on what Computer Science is indeed. Is it science, an art, a craft, or an engineering discipline?

It is very difficult to have an answer that satisfies all the people. Clearly despite the word of "science" in its name, many people do not actually believe that Computer Science belong to science in a strict sense otherwise we would not have had such debate. Simply put, as we all have known, Computer Science also has its focus on the creation of new things such as computer hardware and software rather than simply understanding existing things (there is no computer in the natural world). Therefore, Computer Science drives the creation of powerful computer hardware and software that have shaped our life profoundly. This alone suffices to say that Computer Science has its engineering aspects due to this focus.

In addition, the aesthetic aspects in computer science, especially in software development and hardware design, also demonstrate designers' inspiration and creativity. For example, the explosive popularity of Apple's iPod cannot be simply accredited with its superior technology. This makes Computer Science more like an art that requires great craft skills and taste.

With this brief multi-faceted view of Computer Science, we will examine two claims in this article and then make our point. One claim is from Paul Graham who is a famous hacker and writer of the book

"Hackers and Painters" [2]. In Chapter 2 of the book, he claims that hackers are more like painters than scientists or engineers. The second claim comes from our instructor, Dr. Tolbert, who claims that Computer Science is a craft on its way to becoming an engineering discipline. We examine these two claims in detail and conclude that both claims have truth in them. We argue that Computer Science is becoming a rigorous engineering discipline while at present still leaving plenty room for artistic creativity that demands great craft skills. We also describe the criteria to evaluate good programmers or how to become one.

2 Paul Graham and Dr. Tolbert's Claims

In the chapter of "Hackers and Painters" [2], Paul Graham frequently refers to Computer Science as a craft. He compares hackers with painters, scientists and engineers and points out that hackers are more like painters than scientists or engineers. He mostly draws from his own experiences of studying painting in art school and demonstrates the following common aspects shared by hackers and painters:

- Basic theory is needed to do good work.

  A programmer has to understand the core algorithms that he uses to perform some specific task, and painters need to know about the basic chemistry to choose suitable colors for beautiful paintings. That is, some knowledge of basic theory is necessary to perform a better job.

- Learn by examples.

  Painters can learn from masterpieces, and programmers can also learn by reading good and well-organized code. It is true that a lot of practice is inevitable for a good programmer and a craftsman like painter. Practice makes perfect: By doing the same thing over and over again, one can get a lot of hands-on experiences.

- Gradual refinement

  A good piece of code cannot come out of one's mind all of a sudden, To make a good software product, one needs to keep working and thinking about it and find out if there is a better way to do it, and then makes progress gradually.

- Handling small details.

  Like craftman, a good programmer has to pay special attention to the small details in his code. These small details can also be non-trivial.

First, we would argue that Graham believes this because of his background in painting. People with prior experiences in other fields have the tendency to deduce similarities when looking at Computer Science. For example, people from mathematics background may place more emphasis on mathematical aspects of Computer Science while people from Electrical Engineering stress on its engineering aspects.

If we examine Graham's claim more carefully, we can see that these similarities also apply to engineers. Engineers also need these skills to perform their jobs competently. We cannot agree with Graham's view that engineers just implement the specification with any design. Graham's experience in some big companies left him the impression of no need for design from engineers which are simply not true. Many software engineers still lead the design process and then conduct the implementation. Managers may decide what features a software product should include, but usually they simply do not have the competency nor the necessity to dictate the design. In this aspect, we would argue that Graham's view is polarized and many software engineers are indeed involved in significant design work.

However, we cannot simply dismiss Graham's arguments because they indeed apply to a small group of people: Great hackers. If we classify software engineers roughly into two categories: Great hackers and good hackers (programmers), then we can see that they require different sets of skills.[1] Great craft skills are necessary for great hackers, but not mandatory for good programmers. On the other hand, craft skills alone are insufficient to satisfy the ever-increasing demand for high quality software that needs to be delivered in a timely and economical fashion.

Our argument is that Computer Science requires both great hackers and good programmers with the latter being the majority and that Computer Science is becoming a rigorous engineering discipline for the majority of the people while still leaving plenty room for greater hackers to exercise their creativity and craft skills. In this sense, we are in agreement with Dr. Tolbert's claim that Computer Science is a craft on its way to becoming an engineering discipline.

Following we will discuss the dueling artistic and engineering aspects of Computer Science that has aroused so many discussions and debate.

3 Dueling aspects of Computer Science

As we have stated earlier, the pitfall in Graham's argument is that he does not differentiate between great hackers and good programmers. Great hackers and good programmers require different skill sets even though these skill sets are overlapped.

In this section, we first focus on the engineering aspects of Computer Science that make it different from art and classify most software engineers into the "good engineers" group rather than "good painters". Then we will discuss why Computer Science still leaves room for creativity and artistic design.

- **Great craft skills are not universally required.**

  Design and implementation of certain software, say operating system, are hard. We cannot deny that that requires great craft skills. People who can do that competently are deemed great hackers. However, not many people will be involved in OS design. Actually, most people in computer industry work probably on much higher level, for example, developing various applications based on existing operating systems. Some people even only handle some operational issues, such as network and system administration.

  Some of these tasks may be difficult but relatively straightforward. With proper training and experience accumulating, people can acquire the necessary skills to perform these jobs at least adequately.

  Even for some positions that require working at much lower level, for example, device driver development, that is not too difficult to master. As long as the developer understands the interface between the OS and the device driver and the hardware specification, he will be able to implement just exactly as the specification says. The work may be tedious but does not need much original thinking and artistic skills.

  During the early stage of device driver development, there may be a relatively long learning curve to overcome because it needs in-depth understanding of how a given platform functions, both at the hardware and the software level. However, after years of experience in this field, common knowledge has been developed and people new to this field only need to study and follow the common knowledge to accomplish this task.

[1] Here we do not consider "poor" programmers to facilitate the discussions. Later on we also state only "good programmers" or "good engineers" instead of "good hackers" to avoid the hackerish denotation.

Nowadays, there are also many software positions that not longer involve creating software. For example, some daily job, such as network and system administration, requires at most writing scripts to combine existing tools to perform some administrative tasks.

- **Good software engineers need skills other than great craft.**

  Many nontrivial software development projects require people to work together as a team. Teamwork is very important for the success of software projects. We often see the case if some people cannot go along with the team, then usually they are replaced by other people who can fit in. This is different from painters who usually work alone.

  In addition, we cannot deny the fact that many positions in a software development team are replaceable in some sense. If someone in one position leaves the team, it is possible to hire and train another person to perform the same task even though the overall performance of the team as a whole may be negatively affected for some time. Clearly if great craft is required of software engineers, such replacement will probably never happen as is the case for painters.

  We can say although craft skills are important for great software development, but they are not required in many cases and they alone cannot precisely characterize the nature of many software engineering jobs.

- **Computer Science mostly follows gradual evolutionary path.**

  In painting, whenever new painting style debuts, it always seems to be radically different from previous ones and there is no necessary relationship between a new style and an old one. For example, radical in their time, impressionism is a light, spontaneous manner of painting which began in France to break the picture-making rules of the dominant Academic art [3]. It was a reaction against the restrictions and conventions of Academic art.

  Compared with painting, Computer Science shows a different style of evolution. When looking at the history of software engineering, we can find that programming languages evolves gradually: One new language always tries to adopt good features from existing ones while providing some newer and better features. Java, one of the most prevailing programming languages nowadays, is a derivative of C/C++ language with a simpler syntax. Java is designed to be platform independent and adopts some good features from C/C++ while trying to reduce the complexity and avoid some shortcomings in C/C++: C/C++ language's lack of garbage collection meant that programmers had to manually manage system memory [4]. Java provides a more robust runtime environment and simplified memory management compared with C/C++.

  There may be some "esoteric" languages that are advocated strongly by luminary people such as Paul Graham on LISP [5], however, most software projects are still done in "conventional" mainstream languages due to their extensive library support, broad applicability, readily available runtime environments and other pragmatic considerations.

  Even the successful Linux operating system draws a lot of ideas from Unix. Its implementation may be different, but Linux still maintains very close similarity to the Unix System V and BSD operating systems.

  In summary, radically different ideas in Computer Science are very difficult to go mainstream mostly due to pragmatic reasons.

- **Most software engineers' work can be evaluated objectively.**

Generally speaking, evaluation of a painting tends to be subjective and varies across different people. People with different background and preferences probably can give astonishingly different remarks on the same piece of artwork. Simply put, to appreciate an artwork is not an easy task for laymen.

However it is relatively easy and straightforward to evaluate the quality of software products even by ordinary users even though they may use different criteria. Using the same criteria, we can usually expect to get relatively consistent and objective evaluations from users. For example, people may ask the following questions: Does the software product support the desirable features? How is the response time? Is it stable for intensive use?

Usability and strong practical focus of software products make them significantly different from artwork.

- **Software is welcome for duplication and modification.**

  When a great painting is finished, it cannot be copied easily without losing the original aesthetic appeal. People with great taste for great work seek only the authentic originals, not copy work.

  On the contrary, when software is released, it is possible that people are not satisfied with it or place more requirements. Developers need to modify it or rewrite it from scratch. This is a common engineering practice that always strives to provide better products or services to meet user's requirements.

- **The creation of most software products is subject to time and resource constraints.**

  Painters can work at will, and they are mostly driven by inspiration and creativity. As described in the "Hackers and Painters" chapter, Graham depicts hackers as people who like to work on interesting projects only. This is probably applicable to a handful of people. However, for most software engineers, they cannot just work by inspiration or can work on "interesting" projects only. Instead, many software projects are always subject to time and resource constraints. For software that is mission critical, the development process needs much more control.

- **Most software products' design and implementation need to consider maintenance.**

  There is no such concept as maintenance in painting, and painters do not need to make any modification to their paintings after selling them.

  However, maintenance plays an important role in the life cycle of a software product. Whether the final product is easily maintainable or not becomes one of the most important criteria for a qualified software product. Software maintenance is one phase in the software development life cycle. It involves modification of software product after delivery to correct faults and deficiencies, to enhance and optimize performance and usability, or to adapt the product to a modified environment.

  Software maintenance is inevitable because of several reasons. First the original requirements for a software product will change. Second, software products need to be adjusted and adapted to changes in the environment or changes from other parts of the system. Third, it is possible to find some errors and defects during field usage, although undiscovered during system validation. Because it is almost impossible to produce systems of any size which do not need to be maintained, good software products should be designed and implemented so that maintenance are minimized and relatively easy.

  This is an engineering aspect that the software product needs constant maintenance after shipping. Although it is subject to debate on how to design and implement software to reduce maintenance load, it is generally accepted that a more formal process is needed to ensure software's maintainability. It can never be an afterthought.

So far, we have discussed in detail about the engineering aspects of Computer Science. We can say that great craft skills are not mandatory for good programmers. However, they do need to have an extra set of skills that are usually found in engineering disciplines to perform their jobs competently, for example, proper time and resource management, teamwork and following proper process in design and implementation.

However, here we need to stress the other part of our argument: Computer Science has its artistic aspect that leaves room for great hackers to exercise their creativity and craft skills. Simply put, great software cannot be created by good programmers alone no matter how many they are. Great software can only be created by great/star programmers and then probably guided by them and contributed by many other good programmers. Now we elaborate on the details.

- **"No Silver Bullet" in Software Engineering**

  In his famous article "No Silver Bullet: Essence and Accidents of Software Engineering", Brooks [6] states that software is inherent difficult because of its essential complexity: complexity, conformity, changeability, and invisibility. Despite the advancement of IDEs (Integrated Development Environments), higher-level programming languages and many others, they only help to reduce accidental complexity. There is "No Silver Bullet" in software engineering that can solve the chronic problem that many software projects fail.

- **Many great software products are designed by great programmers.**

  The history of operating systems has many examples that show how brilliant individuals' craft skills can influence the field profoundly. For example, Ken Thompson and Dennis Ritchie designed and implemented the initial Unix operating system as their "small" project to criticize the then overly general and bloated Multics system. Linus Torvalds designed and implemented the initial Linux kernel inspired by Minix, an instructional OS.

  In the history of programming languages, we have also seen that many of the successful ones are designed by individuals, for example, C by Dennis Ritchie and Brian Kernighan, C++ by Bjarne Stroustrup, Perl by Larry Wall and Python by Guido van Rossum. We observe that successful programming languages are seldom designed by a committee.

  It seems that operating systems and programming languages are two of the most fruitful fields for great hackers to exercise their creativity and craft skills. After all, programmers use programming languages to express their thoughts and solutions to various problems and depend on operating system to provide the critical services they can use. In a few cases, creators of these programming languages or operating systems have become the Benevolent Dictators for Life (BDFLs) [7].

- **Programmers vary a lot in productivity.**

  Many people have observed that programmers vary a lot in productivity, for example, in Brooks' book "The Mythical Man-Month" [8] and Graham's article "Great Hackers" [9].

  It is clear that programmers do not play the same role in software development projects. As suggested by Brooks [8], a "surgical" team is formed with good programmers addressing critical architecture design and system implementation with other programmers playing supportive roles.

  Good programmers' taste and craft skills can largely dictate the success of these projects. Actually they need to consider the time and resource limits and oftentimes the best implementation is not the most "suitable" implementation for the target product. All these still require engineering knowledge and practices.

In recent years, we have witnessed the success of many high-profile open-source software projects, for example, Perl and Python programming languages, Linux and FreeBSD operating systems, Apache Web server, Firefox Web browser, MySQL databases and many others. However, we cannot simply give all the credits to their initial designers and implementers because many other good programmers contribute and sustain their successes.

Let us use Linux as an example. Currently Linux runs on many platforms and has many advanced features that are not even available in commercial OSes. Obviously Linux Torvalds alone cannot personally make all the changes. Even he alone cannot accept all the patches to the source. Instead he relies on a core team to scrutinize the patches and then submit suitable ones to him. The core team also depends on other contributors. Other software projects follow a similar hierarchical structure to manage the ever-increasing complex process. Here we observe both artistic and engineering aspects.

4 Evaluation of Good Programmers or How to Become One

Like painters, great programmers are evaluated by the work they done. As we have discussed earlier, most criteria in evaluating software engineers are quite objective and knowing them may help aspiring software engineers to learn how to practise and become one. This is very practical for software engineers to survive in the competition which becomes more and more intense.

We propose the following criteria to evaluate good programmers with some suggestion from Bram Cohen [10], creator of the most popular peer-to-peer (P2P) file sharing system BitTorrent [11].

- **Strong raw coding skills**

  First, the basic ingredient for the success of a good programmer is coding skill. A good programmer can write effective, efficient, concise and understandable programs. He can always organize his code to reduce the possibility to modify the code in more than one module when there is any change in requirement. Keep software designs as simple as possible while meeting project requirements. One good way to improve code skills is through extensive practice, and to learn from past experience and examples.

- **Ability to build architecture**

  One cannot become an excellent programmer without good coding skills. However, coding skill itself is far from enough for a great programmer. What truly separates the great programmers from the journeyman programmers is the ability to build software architecture [10]. A software architect leads a team, chooses the right tools and designs software in the right way. He should be able to understand clients' needs and resources, designs architectural approaches, creates models and components and evaluate feasibility, creates documents for interface specification, and validate the architecture against requirements and assumptions.

  A good software architecture design can make great difference. For example, there are numerous P2P clients available nowadays, such as KaZaA, GNUtella, EDonkey and BitTorrent. Among all these file-sharing applications, BitTorrent seems to be the most popular one that has evolved to account for the majority of P2P traffic on the Internet as indicated by some measurements of the Internet backbone traffic recently. Some people believe that the novel architecture design is the key strength for BitTorrent's success, and the architecture of BitTorrent is very easy to understand. However, the real difficulty in coming up with a novel architecture like BitTorrent is that it involves fundamentally rethinking all of the basic approaches.

- **Passion and enthusiasm**

  Another way to differentiate good programmers from mere programmers is passion and enthusiasm. A good programmer is usually self-driven and passionate about technology. He is highly motivated by the desire to create artifacts that are beautiful and useful. Staying motivated has practical and important consequences: It boosts productivity.

- **Know about customers' requirements**

  Although writing good computer programs is important and takes great intelligence and skills. One of the hardest parts of a software project is dealing with one's customers and coworkers.

  Analysis of clients' requirements is a key factor to a successful software project development. Taking time at the beginning of a project to define and formulate requirements from clients help software developers to ensure their product to fulfill project objectives throughout the whole process. But it is hard to define software requirements precisely and accurately. There are several possible reasons for this. First, clients may not know exactly what they need. Second, clients do not know how to describe what they need effectively, because they are not necessary familiar with the technical stuff, even they know what the desired features are. Therefore, a good programmer should help clients to explain clearly to ease the process of defining and formulating requirements, help to give them what they really need, not what they say they need.

- **Teamwork**

  For large-scale systems, it is simply impossible to be done by a few people. Most software projects require a lot of people to work together. Therefore the ability to work with other team members collaboratively is an important skill that a good programmer should have. Compared with a good programmer who adds value to the work of everyone in the team, a brilliant programmer who cannot cooperate with others is probably less valuable in the long run. The inability to work well with others would even affect some bright people's career path because of the higher probability of being excluded from decision-making at higher levels. Communication, exchange of ideas and cooperation are much more important than being smart for a good team. Good teamwork skills include effective communication with other people, write good documentations, good interpersonal skills, asking for reviews and taking criticism gracefully etc.

5 Conclusion

In this article, we first present the multi-faceted view of Computer Science that makes it difficult to state whether it is an art, science or engineering. We then examine both Paul Graham's claim that hackers and painters have a lot in common and Dr. Tolbert's claim that Computer Science is a craft on its way to becoming an engineering discipline. We argue that the similarities between hackers and painters also apply to engineers. We then show the dueling engineering and artistic aspects of Computer Science and argue that we need both great hackers and good programmers with the latter being the majority to satisfy the ever increasing demand for them to create software products. At last we present some criteria to evaluate good programmers.

Our conclusion is that Computer Science is becoming a rigorous engineering discipline while still leaving plenty room for artistic creativity that demands great craft skills. This is probably very similar to Dr. Tolbert's claim despite a slightly different focus.

References

[1] "High Performance Computing and Communications: Technology for the National Information Infrastructure Glossary." http://www.nitrd.gov/pubs/bluebooks/1995/section.5.html, 1995.

[2] P. Graham, *Hackers and Painters*. Sebastopol, CA, U.S.A.: O'Reilly, May 2004.

[3] "Art Cyclopedia: Impressionism." http://www.artcyclopedia.com/history/impressionism.html.

[4] "Wikipedia: The Jave Programming Language." http://en.wikipedia.org/wiki/Java_programming_language.

[5] P. Graham, "If LISP is So Great." http://www.paulgraham.com/iflisp.html, May 2003.

[6] Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer Magazine*, Apr. 1987.

[7] "Wikipedia: Benevolent Dictator for Life." http://en.wikipedia.org/wiki/BDFL.

[8] Frederick P. Brooks, Jr., "The Mythical Man-Month." http://en.wikipedia.org/wiki/The_Mythical_Man-Month, 1975.

[9] P. Graham, "Great Hackers." http://www.paulgraham.com/gh.html, July 2004.

[10] B. Cohen, "Great Programmers." http://bramcohen.livejournal.com/4563.html, 2006.

[11] B. Cohen, "BitTorrent Introduction." http://www.bittorrent.com/introduction.html.