2004

Code

Konstantinos Batalamas

//code

By

Konstantinos Batalamas

A thesis submitted in partial fulfillment
of the requirements for the degree of

Masters of Fine Arts

Rochester Institute of Technology
School of Photographic Arts and Sciences

Spring 2004

Approved by

Angela Kelly 5/18/04

Chair: Angela Kelly — Date
Professor, School of Photographic Arts and Sciences.

Elaine O'Neil 5/18/04

Elaine O'Neil — Date
Professor, School of Photographic Arts and Sciences.

J. Weiss 5.18.04

Jeff Weiss — Date
Professor, School of Photographic Arts and Sciences

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: "Code"

Name of author: KONSTANTINOS BATALAMAS
Degree: MASTER OF FINE ARTS
Program: IMAGING ARTS: PHOTOGRAPHY (MFA)
College: IMAGING ARTS & SCIENCES - PHOTOGRAPHY

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, Konstantinos Batalamas, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: Konstantinos Batalamas Date: 06/07/2004

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

For my Parents

Eleftheria and Nicolaos Batalamas

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to the Chair of my Thesis Committee, Angela Kelly, for her continuous support and faith to my work, as well as Committee members Professor Jeff Weiss and Professor Elaine O'Neil for their insights and the creative discussions we had during my academic and professional years.

I would like also to thank my good friend Isaac Rivera for all the motivating and inspiring arguments we had over all those years working on numerous of applications. My work wouldn't have been possible without his technical expertise.

And last but not least I would like to express my gratitude to all my partners, managers and directors at HBO for giving me the opportunity to work on all those great projects. The dynamic working environment that we have developed has been a constant inspiration for my work.

INTRODUCTION

On June 6, 2000 around 5 p.m. a producer stopped by my desk and asked me to show him a proposal that I had earlier that day presented in a production meeting. The proposal was about rebuilding an old site that the company, which had hired me a few months ago, was producing every year. The purpose of the site was primarily to support the content of a successful documentary TV series that was scheduled to air three months after its initial broadcast.

It was a time when online content was starting to take off, and starting to get visually more interesting than in the previous years. Products, and by this I mean not only merchandise that can be ordered and delivered on line but anything that is exposed on a web page, was starting to get a better image a more thoughtful representation. It was an interesting moment to see the producer's reaction as his interest for enhanced content gradually increased, as we were looking at sites on my computer's screen that shared these new technological potentialities. The pleasure of showing and sharing this knowledge of constructing such applications, however, very soon was followed by real disappointment. In my attempts to explain and demonstrate to him, through some of my case studies, the mechanism that brings all these audio-visual elements together to harmonic completion, I realized that the producer did not express much interest or understanding, or even make much effort to understand the mechanism. I found that the producer – a creative person in charge of the look and feel of online projects – was surprisingly unable to appreciate the beauty involved in the process of developing such applications, perhaps the most creative of all parts on those online projects.

A month later, in a meeting with a technical manager from the Information Technology ("IT") department, we were trying to define what IT commonly refers to as the "technical specifications" of the application. This terminology is limited, however, and does not encompass the human factor, which, under my approach, is to define barriers to the developer's imagination and creativity. For the rest of that day and for as long as our meeting lasted in and out the conference room I was bombarded with terms such as programming efficiency, bug-free applications, and meaningless deadlines. The IT managers were more concerned about the number of code lines being written for the application, than with the content of those lines. Everyone accepted without question that the success of the project lay in strictly following the "technical specifications" that were established, even though these specifications were established without taking into account programming content, which is the author's view.

In this thesis, I intend to examine how programming expresses the programmer's vision. This thesis does not intend to examine ways to make programming more efficient, to study theories for bug-free applications or to minimize the number lines necessary for coding a specific application. This thesis will not offer programming tips to help a developer to speed up the rendering process of his code, but will take a closer look at the activity of programming, and more specifically at the ways in which programmers relate to they code they write.

Today programmers welcome any tools that help them to complete their projects more quickly, because the majority of programmers accept that programming is something strictly instrumental, and that the only important aspect of the whole programming effort is whether the application runs with as few bugs as possible at minimal cost. But programming is more than that. After five years working with experienced developers/technical producers in the field and having developed programs for a large diversity of online applications – small or big, simple or complex, interesting or boring, smart or silly – I can see that programmers' relationships to the code they write are parallel to relationship of artists to the artwork they create. Although the obvious goal of a programming application is to fulfill the expectations of managers, directors and users, this is only one aspect of the programmer's relationship to the code, the same way that an artist deals with the art dealer, critics, and art market. Their code reveals a number of things about them. These things do not include only their skills, their talent for mastering a computer language. Their code shows their preferences, their beliefs of what can be done in software or what other things a program can mean.

I have found in many cases that authors of application programs do not care about completing their work in the most efficient way, or implementing systems that will result better, lighter, or faster downloading applications. They disregard marketing warnings for narrowband Internet connections, or they coincidentally "forget" the popularity of the technology they are using. They constantly fail to maintain consistency in the work they produce, even in applications in which they share common templates or systems.

Ironically, they ignore guidelines and technical specifications that in most cases are defined by them to help themselves or others in the field. It is an activity full of surprises, that aside from resulting in the creation of the expected application, also reveals another aspect of this activity, that has more to do with the programmer/creator and his human attribute than with the instrumental and purely practical value of his code.

I will not deny the value of all those systems that have been developed over the years of computer programming to improve the application's efficiency. Nor will I discuss here ways of improving those systems or suggest a different methodology for designing a technically better web page or developing more efficient programming tools. I will not argue that those systems are in fundamentally wrong, but I would be very careful in making assumptions when discussing anything that concerns their utility. On the other hand, I will explore the human aspects of the activity of programming and how human fantasy plays its unique role in as dry an environment as that of computing logic.

A programmer, when not at her desk working on a specific application or not struggling to solve a logistical problem or meet a deadline, still thinks and behaves as she does when practicing on her computer. An online visit to some technical forums where developers meet to discuss, exchange, manifest or propagandize their passionate visions about programming shows something beyond the apparent practicality of programming. The discussion can range from a logistical problem that a developer had the night before working on her project to the irritation that someone from the chat room might have had by reading someone's code. The motivation to define a topic for discussion in these online chats can be something as simple as an arithmetic malfunction that returns a value that it shouldn't and delays a programmer's creation from its completion. It can also be

complicated and difficult to analyze in those chat rooms topics such as how to write a truly useful application, or how to develop software that can make you a successful top selling programmer. Whatever the case, the subject does not always remain the same throughout the argument. It often leads to unexpected, unpredictable areas of human thinking where the argument is no longer the same as it was when at first defined, but is more about the person that argues anonymously behind the screen name. It is more about her desire to create something using the programming environment and to express herself than to pursue an answer to her initial technical question.

But why would someone, particularly an art student, who hasn't follow the path of an ordinary software professional or even that of a computer geek be interested in programming? Why would she devote all her time to playing with different computer programs, messing with the structure of an application, or recoding a program that works already? Are there no other more visually interesting and more direct avenues of personal expression for that artist to seek, experiment, and enjoy than to spend countless hours attempting to understand the basic concept of 'variables' in programming? Is it a masochistic dysfunction of the thinking process or a disappointment from the outcome of her effort to produce art that looks and functions with the one prototype designed primarily in her mind?

Important factors of the everyday nowadays rely almost totally on the correct functioning of programming systems. You use your cell phone to make a phone call, your ATM card to withdraw money, a microwave to warm a glass of milk, and a computer to write text or make art. What we call today a postmodern era we could easily name it even if it sounds a little awkward in the software-operated era. From a simple

electronic device to a complex multi-functional mechanism, there is almost always one or a group of microprocessors to operate them. These systems have to be programmed by hundreds of thousands of developers/programmers whose ritualistic thinking and practices change our daily routines. The experience of writing a program is, at the end of the day, what originates all those systems.

But before progressing any further, allow me to introduce the most real aspect common to the majority of the programmers that are involved in development on all the systems mentioned above and many more. The completion of a product such as a web application or a device run on a microprocessor that operates based on a program that may have been written months before the product's final completion doesn't necessarily mean the end of the programmer's work on the project. In fact it means the beginning of what I would call the *second phase* of the project, which phase may last even longer than the development of the program. It can be more frustrating, but it can lead to very unexpected and fruitful areas. It is the part designated by the production team as the *testing* or *troubleshooting area.*

Here are a few examples of how a peaceful and productive day at the office can be seriously disturbed. Someone playing obsessively one morning with the product finds a "bug" in a feature in which no one was expecting, or a new functionality needs to be incorporated into the application that hadn't been mentioned previously or that was accidentally left out of the manager's agenda. The story continues to more dramatic developments. Someone from the team deleted a line of code that it shouldn't have touched or, even worse, someone had the "brilliant" idea to change a function of the code without *comment.* These actions are in the everyday menu of a programmer's life. They

are actions that she can affect other developer's work as well. The same ideas that a programmer may find to be of great value, some other programmer may find to be worthless or ugly and so she may delete, modify, disregard or criticize them to other colleagues to try and convince them of how wrong they are.

This programming activity, the relationship of the author to the code can be studied from two different directions: the '*public*' and the '*private.*' Although they are equally important for the project, in my work I am only interested in the *private* one. However, I would like at this point to give a short description of the *public* approach of the programming activity so that it can help you later to distinguish the difference between the two of them.

From the *public* perspective, programming consists of measuring, calculating and applying successful formulas to achieve solutions on a current application. These calculations must always follow an objective methodology according to which programmers must leave out of their work any chance for personal expression. Under this approach programming is just a flat industrial activity like any other which exist only to achieve a result: the completion of a product – for example, a web application in which the actual *doing* is totally meaningless. To that extent, programming became a soul-less, dry activity that no one needs to be aware of or needs to care about, an activity that should be forgotten and should disappear after its completion. The process of programming is not interesting to watch; only the programming results are. Actually not even the results, but only the utility of those results are significant. The results themselves are totally uninteresting. This public, utilitarian perspective of programming diminishes – if not eliminates – the relationship between the author and her creation

between the programmer and her code to a functional step in the overall production of the application. The arguments for this attitude are the same ones I have experienced since I started working as a programmer: time, money, efficiency.

This perspective is based on the assumption that programming is purely an objective activity. Software journals are dedicated to the publications specialized to the programming as a tool that needs a constant optimization. They publish articles that measure all the factors that make a program more efficient or they examine different modules developed by the experts that achieve the above goal. It is perfectly reasonable under this approach to elaborate suggestions for making the programming effort more efficient, more predictable and more controllable. The question is not whether such approach is legitimate, but whether one can expect it to be fruitful.

The programming activity contains in its nature an erratic side, which has been generated entirely by the programmer's persona. This erratic side can be easily noticed even when you look at the whole phenomenon of programming from its 'public' approach. The best examples of programming I have seen – and in my case the best of my applications – are not the most 'efficient' ones, and by efficient I mean how the term is given by the *public* view: programs that function to achieve only a desired result without wasting a line of code for intrinsic expression. The more efficient an application is does not mean it is better. The more economical applications are not the most interesting, and the ones delivered on time are not the most personal. The best of the cases I've seen so far are those that are not finished in time; they arrived late and cost more than they initially planned. There might be an explanation of this oxymoronic scheme: how an activity which is all about competence and efficiency can be so

unavoidable inefficient. Frederick Brooks in *The Mythical Man-Month* published in 1995 discusses why 'programming is so difficult.' He argues that programming development has a special nature that is both perfectly logical and perfectly immaterial – an activity that we haven't yet learned how to master.

Despite the effort and the detailed analysis of some of the aspects of programming, we don't see in the book the possibility that this erratic nature of programs can originate in the personal relationship that programmers establish with their code. In other words, Brooks does not examine the possibility that the projects may arrive late not because there was not much work put into them, but because the programmer works far too hard on it. Something that has not been mentioned is that programming for programmers is the act of creation and the code is a mirror of them. The code speaks about them as programmers/creators. This is not something that cannot be done in haste, and it is something that can not be timely appreciated by those that do not understand the magic of their creations but care about them as persons. These are the *private* aspects of programming.

From the *private* perspective the act of programming is not something unimportant that happens only to achieve specific results to the application. It is the foundation that defines the entire development process. It is the engine that coordinates and supplies data to the system. It is the heart of the project where unprocessed and meaningless bits of data become useful information. The program itself, from the author's view, cannot be less personal or less subjective than his own vision and beliefs about programming. Everything else comes secondary. Corporate policies, technical specifications, and marketing obviously all exist and are all important factors for the

course of the application, but they all remain minor to the author that he fanatically stays focused on his code throughout the entire development.

Now this is not entirely true for all developers. Some are totally blind to the intrinsic values of the code and are concerned mostly about the public aspects of the programming, such as its usefulness or its efficiency. There are many among the programmers' community that belong to this category. It is really sad and difficult sometimes to discuss with them programming subjects that communicate ideas on a common logical platform, only because they are driven by nothing that involves their thesis, analysis, or even experience in those matters. But for most of the programmers this is not the reality. For them the code is something very meaningful, something important in itself, regardless of its role in project for which it is been associated. It is an activity that identifies them as creators; therefore, the code should be unique, personal and elegant. The program's role and its use for the project are also important aspects of the *public* approach of the activity, but it is not the only one that measures its worth. Programmers also care about code issues that don't only make the system run, but also speak about their preferences, their likes and dislikes, and their acceptance and disregards different styles, methods, and theories about programming.

What from a distance may look like objective technical decisions, under the *private* speculation of the programming activity are nothing less than subjective choices of a mixture of aesthetic preferences and beliefs. Programming is something very esoteric and we may understand some of its aspects by comparing it to art objects and the process of their creations instead of thinking about it as a matter of plain calculations. And for those that manage those projects, their task is to understand the difference

between solving a computing problem and programming. They have to take into account the programmer's relationship to code.

In a very recent example, we had to define the 'technical requirements' of an application in which the company where I work decides to redesign. In our first meeting with the IT people, we were discussing different scenarios for approaching the very sensitive matter of communication between the front end the back end of the application. It was so clear from the very beginning that whatever direction we choose to define the way in which our application will communicate with the server should be the way that IT is mostly familiar. It contains minimum involvement on their part and adds a level of security for them. They were all different methods that we can follow and achieve the same result for the application. But it turns out to be the programmer's subjective choice, which direction will be finally followed to make this objective decision on this particular problem. My choices for the application do not affect its efficiency. They may have an impact on the time been scheduled for producing the application or its available budget, but they were not made based on the above factors. More specifically my choices were not made based on what makes sense to the IT managers and developers. They were based on purely personal preferences and what from my own experience and involvement in the field I consider as beautiful and interesting in programming. For them my code was a detached activity a sort of calculation carried out without engagement, a process of perfectly objective reasoning. For me it was a form of expression.

It is nearly impossible to explain these concepts to everyone, especially to those who have never been exposed to any of the areas of programming. Nor could those who have had the same or more experience in the field explain in depth the technical particularities of the programming practice. I am not a programmer, and so it would ironic, if not a mistake, to claim such role. So, instead, I will give you an example from my brief experience in the act of programming and give you my own witness to the private aspects of programming.

It was exactly a year ago when I was assigned to a project which we had been talking about in my department for quite some time. It was the redesign of the company's video player, an application that brings together all video content there and throughout the corporate site. The project seemed to many of the developers just to involve any other interactive application that we had created in the past, with nothing terribly complex or challenging, at least from the technical side. As for the interface design, there was nothing new that had to be added to the overall aesthetics of the site in which the application would be hosted. The technical objective was the same as in all other cases belonging to the same family of web applications: to build an environment in which part of our knowledge and technical experience would be applied. As for the production, if the project is cost and time efficient, the rest of the development aspects don't really matter. My tools, and by that I am mean the software I was using, had enough built-in components to help me construct the application in a way that would

satisfy the needs of the product. The authoring environment already included database and gui (graphic user interface) tools advanced enough to create what we needed.

But I chose to proceed another way: I chose to write the application instead of using the components offered by the software I was using. Although the predefined scripts offered by the program were not as versatile as my methodology, I believed that it would serve, my purpose at least to some extent, as well as my own application would. In fact, if I had used the predefined scripts, it may have taken less time, since the software offers many ready- made elements in it. Also the application that I would wrote would neither be noticeably faster nor more efficient than if I had choose to build it using the offering components. The application was relatively small to allow any real difference. Nor did I prefer to program it for economic reasons since I would have to build it in any way we would have chosen.

So why I did prefer to create my own application and program it on that very analytical level? Simply because the application would be finer and much more well-defined than using a template to generate all the necessary elements contained inside the application's program. The computational problem might be more efficiently solved using the available tools, but I was not solving a computational problem; I was structuring information. In other words, I was programming. And engaging in programming allows you to include other elements – such as writing code – about which you are happy: code that talks about you and your thoughts about Information Design and Architecture about Functionality and web Usability; about all those different disciplines that I am dealing with in my everyday routine. Those abstract concepts that perhaps mean nothing to a visual researcher/artist for me were my sources of inspiration. I didn't

feel that using only readily available software tools to produce my application would be subtle enough to let me write what I wanted; not that I necessarily wanted to show the program to the whole world or that the world would be interested in seeing it. One could still say it was a matter of private pride.

The aesthetic aspects of programming are, of course, not limited to liking and disliking programming environments, or preferring one programming language over another. Designing such an application requires familiarity with OOP (Object Oriented Programming) languages and some technical knowledge of databases. But the process of designing a program is not just the *application* of the above – knowledge as if programming was a matter of deduction. Instead it requires the programmer to create something. And I do not mean to imply anything extraordinary in this creation; it only represents the unspectacular notion that there was nothing there has to be in a Code and there are decisions that cannot be deduced but that must nevertheless be made. Things like what sort of object we will need, what are their common and unique behaviors, how they communicate to each other and so on. These decisions are interconnected, of course, and choosing the properties of an object, for instance and choosing those properties or those type of behaviors for an object will influence not only modes of access to them but also the visual and conceptual forms of that object. In such a small application as the one that I was assigned to work on, the relationships are more or less easy to define, but they still need to be designed. Needless to say, there are no formulas to apply and one has to choose from among the different alternatives without the help of a scientific model that calculates the best solution. The program involves, therefore, not only applying one's technical knowledge but also making decisions that cannot be based

on scientific laws: one loop or two, this type of loops or the other, this type of class or that, these properties or those? And so on. There is nothing in the way of a *science* of programming with a formula to tell which solution is most appropriate. The Computing Science has no answers to these kind of questions. The grounds upon which programmers make these technical decisions are varied, and it is not easy to distinguish among them. Programmers may base their decisions on economic rationality, or on aesthetic preferences, or on tradition, or even on beliefs (about the users' needs and expectations, about the economical, functional or political consequences of certain technical strategies, about the longevity of the application, etc.) and it may prove impossible to distinguish one ground from another in real life situations. Real life is much messier than what the result of sharp analysis leads us to believe. In this work, I am interested in what programmers say about programming when they try to explain some programming behavior and not to consider the programming project (and programming in general) as a series of objective choices, therefore an activity that progresses according to a coherent course.

For instance, in many cases during my work as a developer I had to face the legality of every single act in programming. I might be told that "What you are doing there is not legitimate," based on arguments of economical rationality, namely, trying to minimize costs. Yet even this is not as straightforward an approach as it may seem at first. The concept of "minimizing costs" is, on the contrary, a rather thorny one, since it is not clear what makes a program cheaper. Is it cheaper if it is written faster, or if it can be used for a long time? Or if it contains fewer bugs (and requires less maintenance)? Or if the code can be written in such ways that could be expanded or adjusted in the future?

It is difficult, and perhaps almost impossible, I would say, to find final answers to these questions, for reasons that could well be explained as bounded rationality: programmers do not have access to all the information needed. In fact, it may be impossible to gather all the information needed, not because of errors in the search, but because of the intrinsic properties of the world, or more specifically, of language. In other words, it is not our laziness, and it is not that we do a deficient job of finding all the information needed to calculate the cheapest option; no, in fact, it is impossible to obtain all that information.

I believe that when programmers make decisions based on certain principles – even when on non-private principles like the minimization of costs or the development of a more efficient application – their reasoning is not based on comprehensive calculations, but on assumptions that originate in experience: hearsay, beliefs and other things that cannot be labeled Computing *Science*. Hence, even when programmers argue in economic terms, the private aspects of programming (their beliefs and experience) have effects on their applications.

In some cases, the economic discourse is not even raised, perhaps because of a clear insight into its flaws, or lack of interest or simply the programmer's criteria of addressing those issues are based on a very personal perceptive behavior of appreciating and understanding the world (and, more specifically, that of programming). Another usual way to reason about the technical decisions (about programming, indeed) is along the lines of functionality. The functional discourse centers on the usefulness of software: programmers will explain that they program with the purpose of creating useful applications. Functional rationality is, however also flawed, and in a similar way to economic rationality: programmers cannot possibly know exactly how their technical

decisions will affect the usefulness of the program. This, naturally, does not prevent them from making decisions according to what *they think* will be more useful to the customer. Another difficulty with this line of argument about technical decisions is that often the alternatives presented to the programmer make no difference at all to the customer, since their problems can be solved satisfactorily in a number of ways. How then can a programmer make decisions based upon the needs of the user? Yet another difficulty is that programmers may, for a number of reasons, misunderstand the customers' needs completely: they may be uninterested, or tired, or may lack the capacity to place themselves in the position of the users, or the users may lack the capacity to explain their needs, etc. In some cases, the discourse does not include any advanced legitimizing maneuvers; programmers simply decide they want to use this programming language or this coding style instead of another because that is what they have always done, or because the company requires them to do so. In other cases, there will be direct reference to personal preferences ("I do not like to work with JavaScript"), to political convictions ("never Microsoft") or even to aesthetic opinions ("Perl is ugly").

In my work I am interested in the aforementioned analytical mess that appears when studying what programmers say about that creative activity that is programming. They explain how they program, and what they think about programming and programs, with the help of all kinds of notions: economics, functionality, aesthetics, convictions, all of them relying upon each other ("application X is more beautiful because it is more efficient" kind of statements)... it's turtles all the way round. But the mess does not appear clearly unless one reads the programmers closely, or unless one is lucky enough to hear them discussing programming itself, as opposed to the uses of a given application or

some technical details in an operating system. I was lucky; I found two discussions, presented in the chapter on method, in which programmers exchanged opinions about the aesthetics of software, giving me an inroad into the private aspects of programming. So I suggest that next we prepare our visit to the disordered world of the private aspects of programming through an analysis of the concept of instrumental goodness, i.e. the notion by which we designate what constitutes a "good program."

In this chapter I would like to examine closely two different values of programming so we may better understand the private aspects of programming. The best way to examine the importance of the intrinsic (private) goodness in programming is by comparing it with its opposite – instrumental goodness – which is one of the public qualities of programming. These two are related to each other in complex ways, which can be clearly seen in the ambiguity of some of the programmers' commentaries.

It is indeed possible to imagine that programmers speak either about the beauty of code or about its utility – that is, about its private or its public aspects. The problem is that, as mentioned earlier, the programmers' comments are sometimes ambiguous and it is difficult to know exactly whether they are referring to beauty or to utility. Furthermore, sometimes the programmers are not in a position from which to evaluate the utility of a program, since they are not the final users. What happens then, from an analytical perspective, when programmers say that a program is useful even if they cannot know if it is or not? This class of affirmations gives rise to what I would name semi-instrumental goodness (it could also be called semi-intrinsic), and the mechanisms behind it play an important role in the creation and maintenance of the private aspects of programming. But let us start, as usual, from the beginning.

Esoteric goodness. MacIntyre (MacIntyre 1985) tells the story of a young girl who was tricked into playing chess by an older relative. He promised her candies for every match she won, and made sure he lost regularly. At this stage, it maybe be assumed that the girl played chess because of what victory brought to her (candies), and

not because she particularly enjoyed it. With time and practice, however, MacIntyre tells how she started to actually enjoy playing, and even ceased cheating (to cheat was fine so long as winning was the essential). She had reached a position where she was capable of appreciating the intrinsic qualities of chess; she played chess for the pleasure of it, as opposed to for the promise of prizes.

A similar case can be made for programmers, but not based on the idea that they would program just for the pleasure of it. Some of them clearly do, but their efforts are only the most radical manifestation of the private aspects of programming. The basic element of those private aspects is instead the appreciation of the intrinsic qualities of code, and the relationship that such an appreciation creates between the programmer and her code.

How can you know whether someone feels an affinity towards something? Only by looking at what they do and what they say (assuming, of course, that they do not lie). Programmers, for instance speak about their code; they engage in disputes about what is the best programming language, they write their programs according to programming styles, they defend their personal preferences, etc. Their code, and this is what the empirical material is proving, speaks of them as to their preferences, their skills and their assumptions. This is the substance of their relationship with code, but, strictly speaking, it does not imply that there exists (or that they care for) an intrinsic goodness of code.

A concern for the intrinsic goodness of an object is a concern for the value that this object has in itself, regardless of its use. Now, programmers could very well identify with their creations, and, nevertheless, not worry about anything else but their utility. It could be that their only concern was for what the utility of code said about them. Is it

useful? Cheap? Easy and flexible to use? Such a concern would only have to do with the instrumental (public) aspects of programs, leaving little possibility for private aspects. Programmers and the general public (users, managers, investors, etc.) would have exactly the same perspective on code, and this thesis would not exist.

This thesis does, however, exist, and the reason is that programmers do care for other details in their program apart from their utility, cost and ease of use (from here on in, I shall refer to "usefulness"). This does not mean that they do not care about its usefulness, only that they care about some other aspects as well, and that these aspects are not necessarily related to it. These aspects are, in themselves, of no interest for anyone other than the programmers themselves; in fact, they are often totally invisible to non-programmers. This is why I speak about *private* aspects of programming, and this is why it is fruitful to think in terms of intrinsic goodness: we are interested in the fact that programmers care (also) about code itself, regardless of its usefulness.

Perhaps a pertinent question here is: "how much do they care?" Well, they do not all care the same amount, or in the same way, and they do not all formulate their care in the same way. Some care a great deal, and some care very little, but drawing this landscape is what this thesis is about. Let me simply state that intrinsic goodness is a measure of the quality of code in itself, regardless of its usefulness, and that this is the main concept of this work.

Instrumental goodness. Intrinsic goodness contrasts with instrumental goodness, which is a concept that I have borrowed from Georg Henrik Von Wright's *The Varieties of Goodness*. This work is a study of the meanings – uses – of "good," its purpose being to serve as a partial "prolegomena to ethics", not aesthetics. Von Wright says that

"instrumental goodness is mainly attributed to implements, instruments, and tools – such as knives, watches, cars, etc." and that "to attribute instrumental goodness to some thing is *primarily* to say of this thing that *it serves some purpose well.*

An attribution of instrumental goodness *of its kind* to some thing presupposes that there exists some purpose which is, as I shall say, *essentially associated* with the kind and which this thing is thought to serve well." We say that a hammer *is good as a hammer*, meaning that it serves well the purpose essentially associated with hammers (to drive in nails). Now, since programs are tools – i.e. they are used to achieve some end – it is perfectly sensible to speak in terms of a "good" (in the instrumental sense) program.

This goodness, from some different perspectives, is what the public discourse about software deals with. For some, a program is good if it really helps them carry out their jobs, for others if it creates succulent revenues, for others if it helps them open new markets, and so on.

But I am not interested in the views that non-programmers hold of programs. My work is about the private relationship between programmers and their creations, and I shall only deal with other interests marginally. Now programmers also care about the instrumental qualities of programs (including those they write), they are well aware that they play different roles (tools, products, services). Only on very rare occasions will we find programmers totally ignoring the instrumental sides of a given program: a program is practically never written without a goal, just for the pleasure of putting commands together. Programs, so to speak, always have a mission, however small it may be. So it would seem that we are drifting away in our analysis of the instrumental and intrinsic goodness of software. These two are apparently easy to distinguish, and the reader might

be wondering why this issue was brought up at all. Perhaps you could say a program has instrumental qualities, but this thesis is about the private aspects of programming and they both seem unrelated to each other.

Obviously, there is more to say. The world seldom complies with the analysis, and there are always problems. In this case, one of the main problems emerges with the notion of the "essential purpose associated with a tool." What is the essential purpose associated with a program? And is it possible at all to assess how well the program serves this purpose?

So, defining the essential purpose of a program is difficult, but what does it have to do with the private aspects of programming? It has one very important thing to do with it, because it happens that programmers hide private aspects of programming behind a seemingly instrumental discourse, and this phenomenon is most visible in the cases where the essential purpose of a program is difficult to pin down. In order to give a good example, I need to explain a programming concept. The concept of readability for instance. With this, programmers mean that the code can be more or less difficult to interpret, in other words, that it is more or less difficult to see the structure of the program, what each component is used for, why, in what order, and so on. Readable code is easy to understand (and to fix and modify, if need it to be); unreadable code is a pain.

But also, even in this very basic concept of programming, there is an added difficulty when trying to connect it with the cost of the project. Readability is not a homogeneous concept, i.e., programmers do not all agree on what it is that makes code more readable. Some say, for instance, that a lot of comments make it so, others that a lot

of comments clutter the code and make it more difficult to read. Readability is, in other words, a subjective quality of code. It may also be argued that by 'readable' we mean that a program is understood by a majority of programmers, that the concept has a statistical meaning. What we face here are two different uses of the same word. A number of theorists, with a particular approach to software development, use it in the statistical sense, putting aside their own, or any other individual, opinion.

"Readability" is, for them, a characteristic of programming that can be measured statistically, code A is x% readable, so to speak. This can be measured by having a number of programmers read the code and answer questions about it. The validity of these measures is an unresolved question, but that is a problem of the structure of the experiments, not of their concept of readability.

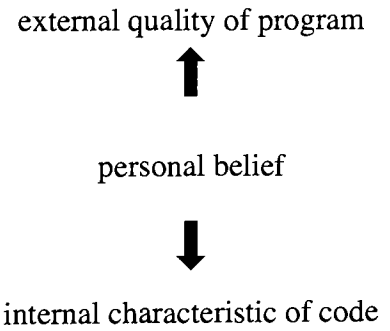
But readability from the programmer's point of view is not something that can be measured statistically. The programmer will always have her own subjective opinion of the readability of a program. The version of readability that I am interested in is the one used by programmers – I am, after all, studying the experience of programming, so I must consider it as a subjective quality of programming. For our purposes, then, reading code is not a statistical phenomenon but an individual one.

Now, this work is an argument against the idea that programming is not only about calculating optimal solutions, it is also, in a deep sense, a matter of personal decisions: decisions that someone can see on the programmer's personal style of coding. This does not mean that some programmers are not better than others. To follow personal beliefs when programming, is not a shortcoming but the only possible way to program.

Well, this last statement is not quite correct. There is another option: to program, according to your personal preferences, and not make any connections between those preferences and the public qualities of program. Some people believe that one should write readable code out of kindness to other programmers, or that she writes readable code because she likes to. Now, presenting one's preferences exclusively from the private perspective is not easy, mainly because it is not legitimate but also because this is not the way programming is generally taught and it is not the way programming is discussed.

At any rate, there are beliefs that connect the internal qualities of a program with the external ones. Those are, I would say, personal beliefs that the programmers have for the instrumental aspects of programming because they are used to making statements about the instrumental properties of a program, when such statements cannot, from a strict perspective, be made.

The role of those personal beliefs can be illustrated with the following figure:



They are often used unconsciously, so to speak, and in rather vague forms. For instance, few programmers will go so far as to say, explicitly, that readable code yields

better tools. Instead, they will say that readable code yields better programs, without defining what kind of goodness they are referring to. In fact, they will more often speak in personal terms, explaining how they think one should program and loosely claiming that programming so yields better software. I have therefore decided to include a third kind of goodness, the vague one that programmers refer to when they express personal beliefs. This is the "semi-instrumental" or "semi-intrinsic" goodness. The analytical value, and elegance, of such a concept is questionable, but I think it points to an important aspect of the personal relationship between programmers and their creations, namely the unclear border between the public and the private discourse. When programmers discuss programming, both kinds of reasoning appear often close to each other: beauty, for instance, is mixed with maintenance, which is directly connected to costs and utility. The private aspects of programming are manifested very clearly in particular phenomena (such as aesthetic discussions, harsh disputes and other details) but also, although less clearly, in their comments about what it is that makes software better.

In this chapter I have shown how programmers use the concept of readability and how this use points to the existence of a hybrid kind of goodness, not strictly instrumental but neither properly intrinsic. Perhaps you could say that it is a mixture of private preferences with instrumental legitimacy.

Now, are there other concepts, apart from readability, that can be used in a similar way? Structure and robustness are both of the same category as readability. They refer to the properties of code, but can be used to indicate a kind of goodness that is vaguely related to the instrumental qualities of the program.

Efficiency is a bit special since it may refer to different things: at the programming level, it has to do with optimum use of memory, or lines of code, or disk-space or something like that. This is a quality of the code itself. But it also points to an external quality, namely that of solving the users' problems in an efficient manner. However, there is no necessary connection between an efficient use of the memory and solving the users' needs. Naturally, these two conditions are not opposed, and in some cases they may actually be directly related.

Functionality is a characteristic of programming rather difficult to define, but paradoxically, it is also one of the most important. The concept of *beauty and functionality* is a study of the relationship between the most obvious private aspect of code and the most obvious public aspect of applications.

I believe it was important to present these concepts to provide analytical support of the private aspects of programming. Without the possibility of choosing between different alternatives (and still obtaining the same result), and expressing personal beliefs or ideas about programming development there would have been no personal relationship to code, only calculating and optimizing.

Bibliography

Experience Design by Nathan Shedroff

Envisioning Information by Edward R. Tufte

Visual Explanations by Edward R. Tufte

Visual Thinking

- *Art and Fear* by David Bayles and Ted Orland

The Mythical Man-Month by Frederick P. Brooks, Jr.

The Philosophy and Practice of Systems Design by Bo Dahiborn LARS MATHIASSEN

Action Script by Colin Moock

The Varieties of Goodness by Georg Henrik Von Wright

Designing Web Usability by Jacob Nielsen

The Existential Pleasures of Engineering by Samuel C. Florman

Flash Web Design – The Art of Motion Graphics by Hillman Curtis

Internet Sites

http://web.media.mit.edu/~minsky/papers/CausalDiversity.html

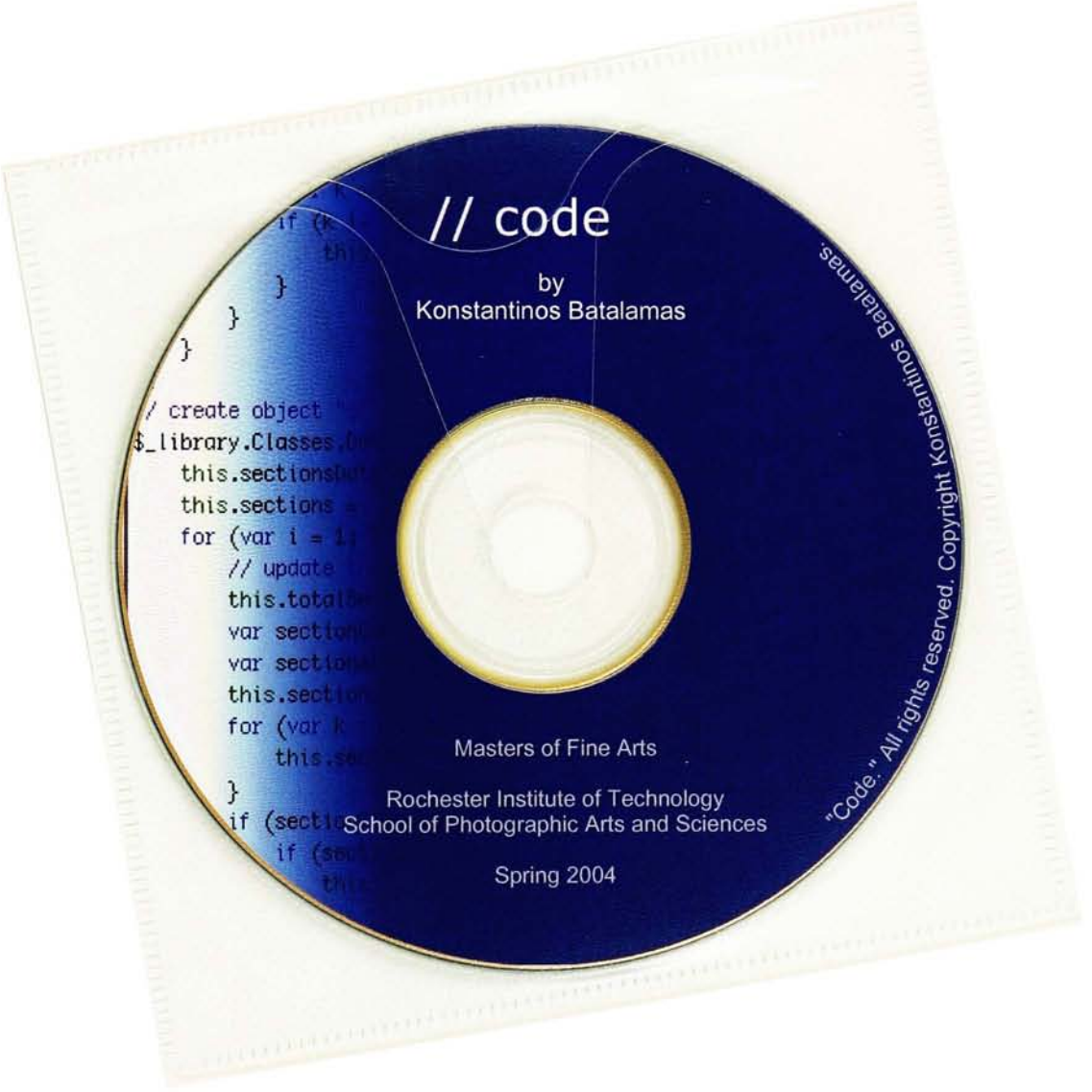
http://www.chc-3.com/pub/beautifulsoftware.htm

http://www.cfmc.com/adamb/writings/goffman.htm

http://www.macromedia.com/cfusion/showcase/index.cfm?promoid=home_sod_022704

Konstantinos Batalamas

Rochester Institute of Technology
School of Photographic Arts and Sciences
58 Lomb Memorial Drive
Rochester, NY 14623-5604
585-475-2229
www.rit.edy


// code
by
Konstantinos Batalamas
Masters of Fine Arts
Rochester Institute of Technology
School of Photographic Arts and Sciences
Spring 2004
"Code." All rights reserved. Copyright Konstantinos Batalamas.