a text, it functions like a machine. Latour (1992) pointedly observes, paraphrasing Austin, that "how to do things with words and then turn words into things is now clear to any programmer." The classical distinction made in engineering between *designing*, i.e., drawing the blueprints, and *building*, i.e., assembling the physical structure, does therefore not translate well into software programming. According to Jack W. Reeves (1992) writing the source code can be compared to designing but building is nothing but the automatic translation of source code into machine language by a compiler program. In contrast to classic (hardware) engineering, software is thus expensive to design – it takes a lot of time to write a functional piece of software– but cheap to build. From an economic perspective, we can even speak of an apparatus of production unlike other areas of technology, specific to the creation of software: except for the price of a computer, producing software is basically free, time becoming the essential cost factor. In this sense, software is again closer to literature or music than to industrial production – the workstation is the factory floor. This greatly facilitates people shifting from consumers to producers.

Like knowledge and information, software can be shared without tangible loss for the giver. The Internet transports and copies computer code as simply as text, sound, or images; algorithms, program libraries, and modules pile up at different sites, contributing to what could be seen as the equivalent of a fully equipped workshop with an unlimited spare parts inventory attached to it, accessible again at the cost only of time and skill. A general-purpose programming language like Java nowadays comes with thousands of ready-made building blocks and writing code is often closer to playing Lego than to the laborious task of manipulating memory registers it used to be.

Unlike the products of industry, a computer program is always tentative, never really finished or "closed". Classic machinery also has to be tended, calibrated, and repaired, but with software the provisional aspect is pushed to the extreme. One mouse click and an entire subsystem can be copied to another program and the output of one piece of software can instantly become the input of another. We do not want to encourage in any way the view that holds that everything digital is fluid, chaotic, and auto-organized, but there remains the fact that this freedom from most physical constraints renders software easier to manipulate and handle than hardware objects. The only constraining factors are time and skill. This relative freedom is one reason for the production of software in practice being so unlike engineering by the book.

3.2 Software Design as Heterogeneous Practice

According to IEEE Standard 610.12, software engineering is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software."[4] The attempt to translate the strategies and methods of

[4] See: http://standards.ieee.org/catalog/olis/se.html

classic engineering into the area of software has never been entirely successful and has been criticized from several directions. We cannot possibly summarize all the different views expressed in this complex and long-standing debate, but there are several main critical positions that can be distinguished.

One argument holds simply that programming is based less on method than on skill, that it is craftsmanship rather than engineering, and that "in spite of the rise of Microsoft and other giant producers, software remains in a large part a craft industry" (Dyson, 1998). The main question for design, then, is not how to find the proper methods but how to acquire the appropriate skills.

Another argument is that software engineering has its place but that specific methods and strategies cannot be directly imported from traditional engineering, because building software is very different from building bridges and houses (Reeves, 1992). Debugging for example should not be treated as a hassle to be eliminated by using mathematical rigor, but as an essential part of creating computer programs.

Finally there are those who believe that software engineers should be supplemented by other professions, in particular by software designers who take inspiration from architects rather than engineers because buildings and software "stand with a foot in two worlds – the world of technology and the world of people and human purposes" (Kapor, 1996). In this view, building a computer program is not so much about technical problems, but about how to bring users and tools together in a meaningful way.

Independent of these different views the empiric observation remains that the practice of creating software rarely resembles the top-down engineering models like the *lifecycle*- or the *waterfall*-model where the process of going from neat requirements to a working program is thought of as an advancing in clear cut stages. The "real world" of software development is most often described as "messy, ad hoc, atheoretical" (Coyne, 1995), as consisting of "bricolage, heuristics, serendipity, and make-do" (Ciborra, 2004), or as the result of "methodological and theoretical anarchism" (Monarch et al., 1997). While this does not automatically make software production "art", as Paul Graham (2003) suggests, we have to accept that the engineering ideal is just that: an ideal. Software production in practice commonly takes paths that go in different ways beyond engineering. Two important factors have to be taken into account: changing problems and increasing complexity.

First, the problems software is expected to be used to solve are becoming more "cultural" and less "technical." If computers were still doing what they did in the 1960s, namely number crunching and data storage, there would probably be no discussion about software engineering or design. With computers now performing semantic and social functions this has changed. Methods like *participatory design* or *end-user development* are now used to try to integrate the fuzziness of specifications for software by integrating future users into the construction process.

Second, the complexity of software is increasing rapidly and this makes it always more difficult to plan a program in every detail before starting to write code. It is often impossible to foresee problems early on and plans and models have to be