Post-industrial-revolution HCI

Colin Johnson
University of Kent
Computing Laboratory
Canterbury, Kent, CT2 7NF
C.G.Johnson@kent.ac.uk

ABSTRACT
This paper argues that computing in its present state is akin to the state of manufacturing prior to the industrial revolution. It is suggested that eventually an industrial revolution will occur in programming through the use of automated program generation tools, which will allow the rapid creation of programs on-the-fly from what-needs-doing descriptions rather than the how-to-do-it descriptions of traditional programming. What would interfaces to computers look like in this context, and how would they aid users in coping with complexity?

INTRODUCTION
The aim of this paper is to explore what Human-Computer Interaction would look like if computing were to enter what we will refer to as a post-industrial-revolution phase. In the first main section of the paper it is argued that computing as we presently understand it is in a state similar to manual industry before the industrial revolution. This concept is defined and explored, and some notions of what post-industrial-revolution computing might look like are outlined. The core notion is that computer programming becomes cheap, rapid and above all *automated*. That is, most programming will be carried out on-the-fly by automated programming systems.

The remainder of this paper then explores the consequences of this for HCI and ways in which humans can cope with complex problems using computer systems. How will the interfaces to computers change is bespoke programs can be created on-the-fly, and what sort of interfaces will be required to interface with those computing systems? How will this affect our understanding of the notion of "context" if we can process that context rapidly? How can we exploit micro-models created by such automated programming methods to put ambient guidance into systems such as data visualisation systems? And might this lead to a world in which computers are "programmed" by statements of what the computer should do rather than how it is to do it? What might interfaces to such systems look like? And what would still be difficult or complex?

COMPUTING IS IN A PRE-INDUSTRIAL-REVOLUTION ERA
This paper is predicated on the notion that computing is currently in a state which can be compared with traditional manufacturing industry before the industrial revolution. In particular, the production of computing artifacts is still carried out in a manner reminiscent of craft rather than industry; many individuals bringing their individual skills and semi-informally acquired knowledge to the careful production of a specialised artifact over many months or years.

WHAT MIGHT POST-INDUSTRIAL-REVOLUTION COMPUTING LOOK LIKE?
The argument that I would like to base the remainder of this paper on is that eventually (it is hard to predict when) computing is likely to mature to a stage where there is an "industrial revolution" of sorts. The most important feature of this stage in the development of the discipline will be that programming will become a day-to-day activity, with programs being generated on-the-fly as needed.

There are a number of reasons why such a change might eventually arise. The first is an increased amount of work in a nascent field which could be called "automatic programming". This encompasses many topics which are currently considered to belong to radically different areas of computing, unified by the aim of developing systems which are able to generate program source code from descriptions of desired behaviour.

One example of a current discipline which may contribute towards this effort is that of *genetic programming* and related techniques. A second discipline, which is currently seen as sitting at an opposite end of computer science, are formal techniques such as refinement which enable the distillation of specifications into executable programs by gradually replacing more abstract descriptions with more concrete ones.

A second form of evidence comes from sub-disciplines where this kind of change has already occurred. One example is in the

area of data analysis, data mining and statistical model building. Only a short while ago such activities were in a pre-industrial-revolution model, with large, complicated models being developed over many months and being treated as delicate things. This retains a place (e.g. in some areas of economic modelling), but the main mode of working in these areas is now interactive on-the-fly model-building using tools such as *Clementine* and *Weka*. These allow and encourage the production of models as and when needed, using automated model-construction tools such as decision-tree induction and neural network modelling. Another example comes in the area known as meta-heuristics, where general problem solving techniques for optimization, classification, et cetera are developed and then specialized to a particular problem. This contrasts with the traditional notion of programming as building up from nothing.

INTERFACES: FROM "HOW" TO "WHAT"?

One view of the development of computer science is that we are eventually looking to move from a view of computing based on telling the computer *how* to do something to telling it *what* to do. This represents a radical shift in how we interface with computers.

What-based interfaces

What might an interface to such a system look like? The canonical science-fictional portrayal of a what-based interface is given by the voice interfaces on TV programmes such as *Star Trek*. Clearly such an interface would require vast advances in terms of language processing capabilities as well as in the automated generation of functionality from descriptions.

If we are describe complex problems to the computer in a what-language rather than a how-language, then we need to appreciate that there are different kinds of problems. For example some problems are naturally defined by *data* rather than specifications, e.g. a pattern recognition problem. The data is not simply some examples of a Platonic ideal form which can be specified; the data *define* the problem. It has been noted by Partridge that often attempts are made to force these problems through what he calls a *specification bottleneck*, where an artificial specification is devised to capture what should be left as data. Some problems are best defined "interactively" e.g. where the computer generates proto-solutions which are chosen by the user then improved upon. How can we develop interfaces which present the user with a natural way of describing problems in each of these forms?

Is what-we-want really what we want?

It is typically assumed that such systems would represent a radical improvement on conventional programming approaches. It is assumed that if we can "just tell the computer what we want" then that represents a vast improvement on having to tell it "all the little details". In terms of software engineering frameworks we miss out the middle of many models, leaping straight from requirements elicitation to deployment (or at least testing; but we hope *that* will be automated, too).

However perhaps there are some potential problems that we are blind to because of our experience in interacting with traditional software:

- Is it really all that easy to come up with a good description of the "what" that we require? Perhaps we only think that this is easy because we think that how-based approaches are difficult and "anything else" must be better.
- Do we lose something of the "interactivity" that we get with how-based systems? Perhaps we are able to realize complex systems in how-based programming because a human programmer is drawing on a mixture of ambient background domain knowledge, "common sense", small-scale reasoning about the problem and the requirements for the solution and *tacit knowledge* about the specific problem and problem solving in general which an automated programming system would not have. Perhaps the mistakes, trials and prototypes that we produce in programming are part of the essence of producing good programs?
- How easy is it to be "complete" in the descriptions that we give in how-programming? If we miss out a requirement, will we notice that in the final program? How can we constrain what kinds of requirements we need to specify (this is essentially the *frame problem* from classical AI)? Do we find it easier to find missing requirements when we are actively engaged in telling the computer what to do, compared with merely providing a list of how statements and letting automated programming do the rest?

It is perhaps sobering to think that at one point the things we now know as (high-level) programming languages were sometimes referred to using terms like "automated program production systems. The assumption was that such languages would make computing almost trivial; instead of having to mess around with all the "little details" of machine/assembly code, "all" the programmer would have to do with these at-the-time new systems was to tell the computer, in some formal approximation to natural language, how to carry out the task at hand. Easy! Well, not so easy as it has turned out; it seems that we reach a rapidly-steepening cliff of conceptual complexity as we try to "merely" give a high-level description of what to do. Nonetheless it is clearly easier than trying to write programs in assembly code.

CONCLUSIONS

Computer programming is presently a clunky, craft-based industry. Eventually automated program-creation tools may give us tools which can generate bespoke programs on demand. This will engender a radical change in the way in which we interact with computers, moving from a how-based approach to a what-based approach? How will this impact upon our interfaces with computers? Will it be the unmixed blessing that it is often assumed?