GOOD COMPUTING: A VIRTUE APPROACH TO COMPUTER ETHICS

**A book under contract with
Jones & Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
http://www.jbpub.com/**

**DRAFT FOR
June Puerto Rico writing retreat**

Chapter 11
Social Frameworks

**Version 1: 01/30/04 by Chuck Huff
Version 1.1: 02/08/04 by Chuck Huff**

Based on writings in www.computingcases.org prepared by Chuck Huff and Bill Frey and class documents from University of Mayaguez, PR, FILO 3185 prepared by Bill Frey

From a mathematical point of view, computing is a theoretical and technical field. Algorithms, data structures, Turing machines, networking protocols, and operating systems should be founded in the mathematical and engineering approaches that have made them possible. Occasionally concern is voiced for issues in human-computer interaction, but the models for this area are primarily cognitive and psycho-physiological. As the Nobel-prize winning computer scientist and psychologist Herb Simon has said, hard science drives out soft science (ref).

We hope the cases and argument in this text will convince you that, though the desire for hard-science rigor and models is a reasonable one, you will not really be able to understand how computers are used in the real world, or why there are ethical issues associated with their use, until you understand a few basic social science ideas about technology. The two most fundamental of these ideas are that technology is *value laden* and that technology is *socially embedded*.

Technology is Value-Laden

What can this claim mean? Are programs like people in that they have values, attitudes, and desires? Not really (though it is an interesting question to ask from the perspective of artificial intelligence). What we mean is that technology (and particularly computing technology) needs to be thought of as a tool that structures the actions of people. If you design a class registration system for a school, you will need to think about how people will use that system, what needs to be done by the system, and your design will need to take those considerations into account. And that means that what you value will influence the way you design the system. But let's start with a simpler technology.

Designers can't avoid assumptions about users

As you read this sentence, you are probably wearing a wristwatch. Most people in western cultures do. Take your wristwatch off and take a fresh look at it. Pretend you are an archeologist who has just discovered this artifact, along with some magazine advertisements showing people wearing and using them. What would you learn about the people by studying the watch? You could get fanciful and speculate on its religious or social status purposes (in fact, the design of clocks was deeply influenced by the need to know the correct time to say prayers (refs), and we know that watches are often a fashion statement).

But at first, let's be more basic than this. What does a wrist watch designer assume about the user when designing the watch? What are the straps for? Clearly, the assumption is that it will be strapped around an appendage of the body, and given the length of the strap, likely an arm. So, it assumes the user has arms.

How do you read a wristwatch? Well, you move your arm so the watch is in sight and look at the dial (or display, if it is digital). So, the watch's design assumes you can move your arms in this manner, and that you can see. Another assumption is also included in the first sentence of this paragraph: that the user can read.

If you keep unpacking the assumptions of the wristwatch (it is a useful class exercise) you can come up with a long list of assumptions its design makes about the user. Some are more abstract (like things the user cares about) and others are more concrete (like whether the user has arms). Some are true of all wristwatches (the thing about arms) and others are only true of some models (like watches with dual time displays). So, some design assumptions will vary from model to model, and as this exercise shows, can often be inferred from a simple, thoughtful analysis of the artifact.

Here is the first step in our argument that technology is value laden: technological artifacts make assumptions about their users in their design. This is not a bad thing. In fact, making explicit one's assumptions about the users of a proposed software product is essential to good design and to good requirements analysis. We have to make assumptions, and we are better off if we are explicit about them.

Systems can contain bias.
But there lies the rub. We are often unaware of the assumptions we are making. Here is an example from an experiment on software design. Huff & Cooper (1986) had teachers create conceptual designs for educational software. Teacher/designers were told that the software should teach the proper use of commas in sentences at a seventh-grade level. They were then given a framework to help them in generating the conceptual design (e.g. What will the theme of the program be (a trip, a conversation, a quiz, a game)? How will the child interact with the program (through typed commands, a menu, a joystick, etc.)?).

But some of the designers were told that their program would only be used by seventh grade boys. Others were told that their program would be used only by seventh grade girls. And a final group was told their program would be used by seventh grade students (gender unspecified). The program designs these different groups produced differed in an interesting way. Programs designed for boys looked like games (with time pressure, eye-hand coordination, and competition most important), while programs for girls looked like "tools" for learning (with conversation and goal-based learning). A representative "boy" game had the students shooting commas into sentences, under time pressure, with a "comma cannon." A representative "girl learning tool" was a simple drill and practice program. Thus we now had the prototypical "boy" and "girl" software profiles. So far, it is unremarkable, though perhaps unfortunate, that the software design followed traditional gender stereotypes.

But programs designed for gender-unspecified students looked just like programs designed explicitly for boys. This means that programs supposedly designed for all students were really only designed with boys in mind. This was despite the fact that 80% of the designers of these programs were female.

So this experiment uncovered an assumption about the user that these designers were making: that the users were male. Others (refs) have complained that this assumption is regularly made in the design of real educational software. There is also evidence that software designed in this manner makes it more difficult for girls to learn some subjects.

So this assumption about the gender of user (and the way the assumption is embedded in the design of the software) can disadvantage some users.

And now we can make the second point in the argument that technology is value-laden: some assumptions about users that are embedded in a technology can disadvantage or even harm some users. This does not, by the way, need to be done intentionally, though occasionally it is. No one designing the comma-teaching software intended to harm girls with their design. Nissenbaum & Friedman (ref) talk about software that unfairly treats some people differently as containing bias. It is important to note, that once the program is designed and implemented, the bias continues to have its effect in the action of the software, regardless of the intention or presence of the designer. And thus it is the software itself that contains the bias.

A more abstract way of saying this is that software can be designed to treat different users differently. Again, this is not a bad thing, put this way. We often need to give different users different privileges (e.g. device access privileges in operating systems). But it is an unavoidable choice: we either make different levels of privilege possible in a system or we do not. And either way we have decided how users of the system ought to be treated.

Design choices are often value choices
A still more abstract way to put this is that the systems that computer professionals design will affect other people. Again, this is a good thing. Who would want to design a system that had no effect on anyone? Why bother? We design computer systems to do things, or really, to help other people do things. Computer systems “instrument human action” (Deborah Johnson ref). That is, they provide instruments, or tools, that help people do things.

For instance, while concentrating on which data structure to use in a class registration system, it is tempting to think that what you are doing is manipulating data. But an equally relevant description is that you are is deciding how to construct a tool that will allow you (or someone else) to manipulate data. And the data itself might be data about other people. So, in choosing a particular data structure, you are shaping a tool that someone else will use to manipulate the data about a third person. And, so, you are doing things that affect people, that structure how they can act.

How do you make these decisions? You usually make them based on some knowledge of what “needs” to be done. A good requirements analysis will have helped you figure out who your users are, what the important tasks are that need to be done, how those tasks should be done, who ought to do them, what resources and tools they need, etc. etc. But the requirements analysis itself involves balancing tradeoffs in all these areas. Wider access to data is not always consistent with better security. More flexibility in an interface often makes an interface more complex and difficult to learn. Which of these is more important? It depends, of course, on who your users are, what they value, and on a host of other issues. Thus, your decisions in the requirements analysis are influenced by tradeoffs in values (flexibility, ease of use, access, security). And as the example with gender in educational software, it is possible for other values (in that case a disregard fir equity in access) to creep into a design without the designer's awareness.

So here we are at the final point in the argument that technology is value-laden. The choices you make in designing software are often not simply technical choices. They are often based on either explicit or implicit tradeoffs in values, or assumptions about the users or the task. And these value choices have effects on the users of the system, and often on other people. This is easy to see in any of the cases we have included in this book.

Making thoughtful value choices is part of good design
At this point in the argument, it may be tempting to find a way that value judgments can be avoided. Is there a way that we can change our system design procedures so that value choices are avoided or minimized? Probably not. And even if it were possible, it would not be desirable. When deciding upon the best balance in trading off flexibility vs. safety in tracking an input device, you are making a value judgment (see the Therac-25 case). When deciding upon the best balance in trading off privacy and openness in a networked system you are making a value judgment (see the design of the Finger command in the Machado case).

In fact, if you look at any system design methodology, you will find that much of the method is based on making value choices explicit and allowing for their resolution to be done thoughtfully and systematically. Some of the values we cover in this book are already a part of traditional design considerations. System safety, for instance, is already a specialized field in engineering.

Some of the values we cover in this book (like equity in access) are ones that have not been a traditional part of system design in the past. But many are becoming so today. In the USA, the Americans with Disabilities Acts is putting increasing pressure on web designers and software designers to take disabilities into account (ref). Similar pressures are on the increase in many European countries (ref).

Privacy is another value whose consideration has been increasing recently. Privacy law in the USA is fragmented and often not well enforced (ref). The situation is different in many European countries, where national data privacy laws are now a standard part of business considerations (refs) and there is a move afoot to produce a European Union-wide data privacy agreement (ref). The close connections between the business market in Europe and the USA will likely begin to force considerations of data privacy on any American companies wanting to do business with Europe.

So, many of the value decisions we cover in this book are now becoming a regular part of the computer design landscape. But even if legal constraints were not pushing you to produce software that was thoughtful and systematic in the value choices it makes, it is simply good professionalism to do so. In the end, it is part of professional responsibility to make these decisions, based on the best judgment available. Many system design methods are constructed to help you do so. Toward the end of this chapter, we will provide you with an overview of some of the more recent methods to help designers explicitly consider issues of value. Chapter 2 takes you in depth into a particular method to inform design by understanding the socio-technical system.

Technology is Socially Embedded

The claim that technology is socially embedded is more than simply saying that computer systems are used by people in organizations. Instead, it is the more radical claim that you can't really understand the computer system until you understand its actual use-in-practice. The real system is what has been called a socio-technical system (refs to Kling, Mumford, Bjiker). And in order to understand the technical bit of the socio-technical system, you need to understand its place in the system, and thus the system itself.

In a moment, we will get to some examples, but it is important to understand why this claim is relevant to those who design and create computer systems.

Why should I care?

If you are in school studying computer science, you are likely spending a great deal of your time understanding the computer itself, how it works, how software is structured, how network protocols are used, how operating systems work, and many other technical subjects. This is important, foundational knowledge for your career. But computer professionals who are working in industry spend only part of their time keeping their knowledge and skills up to date. The bulk of their time is spent using that knowledge and skill to solve problems for their clients, whether those are in-house or outside clients.

And here is why the socio-technical system should matter to you. The real problems in business, finance, machine control, databases, communication, etc. are all embedded in a social system, in some particular organization or social communication. In order to understand the requirements for any system you design to solve the client's problems, you will need to understand at least those parts of the problem that involve the tool you will be designing.

For instance, a proper understanding of the software problem for the Therac-25 radiation therapy machine would have involved knowing something about how machines like it were used on a day to day basis. This would have made the time pressure on machine operators evident along with the difficulties with maintenance of the software and hardware of the machine. But the effect these issues would have on the safety of the machine was overlooked, and as a result, people were injured and some died from radiation overdoses (see chapter 6). The cases section of this book is full of examples of this nature.

So understanding the socio-technical system in which a computing system is embedded is important to the proper design of that computer system. In this section we will try to understand what a socio-technical system is and how it interacts with the technical pieces of the system. In chapter 11 we will review some tools and methods that will help you understand particular socio-technical systems for the purpose of informing the design of a computing system.

What is a socio-technical system?

We are not proposing the term socio-technical system simply because we like big words (though we do). The idea of a socio-technical system (abbreviated as STS) is an intellectual tool to help us recognize patterns in the way technology is used and produced. Identification of these patterns will help us to analyze the ethical issues associated with the technology-and-its-social-system. The socio-technical system, rather than the technology alone, produces the social effects that concern us and structures the ethical and many of the practical issues you will face as a computer professional.

It is by now a truism to say that any single technology can be used in multiple, and sometimes unexpected, ways. But we need to add to this observation that, in each different use, the technology is embedded in a complex set of other technologies, physical surroundings, people, procedures, etc. that together make up the socio-technical system. It is only by understanding this system that we can parse out the ethical and practical issues.

Let's take as an example a relatively simple technology: a set of 10 microcomputers connected to each other by a network. The practical and ethical issues associated with these networked computers will change dramatically depending upon the socio-technical system in which they are embedded. For instance, are the networked computers:

- part of the intake unit of an emergency room,
- a small, public lab at a university
- the computing lab of an elementary school
- a risk analysis office in an insurance firm
- a military supplier testing manufactured parts

The networked computers in each of these different circumstances are part of different socio-technical systems, and the ethical and practical issues arise because of the specific socio-technical system, not because of the computers in isolation. Many of these ethical issues are intimately related, however, to the technology: issues of reliability of the system in the emergency room, data privacy in the insurance company, free speech and misuse in the public university lab. These are not just social systems, they are socio-technical systems, and the issues associated with them are based in the particular combination of technology and social system. It is the technology, embedded in the social system, or together with the social system, that shapes the context in which the particular technology must be considered.

You have divined by now that a socio-technical system is a mixture of people and technology. It is, in fact, a much more complex mixture. Below, we outline many of the items that may be found in an STS. But you should be aware at the beginning that many of the individual items of a socio-technical system are difficult to distinguish from each other because of their close inter-relationships.

Socio-Technical systems include (ref to Kling et al.):

- *Hardware (Mainframes, Workstations, Peripheral, connecting networks).* This is the classic meaning of technology. It is hard to imagine a socio-technical system without some hardware component. In our above examples, the hardware is the microcomputers and their connecting wires, hubs, routers, etc.
- *Software (operating systems, utilities, application programs, specialized code).* It is getting increasingly hard to tell the difference between software and hardware, but we expect that software is likely to be an integral part of any socio-technical system. Software (and by implication, hardware too) often incorporates the social rules and organizational procedures as part of its design (e.g. optimize these parameters rather than those, ask for these data, store the data in these formats, only make them available to these people, etc.). Thus, software can serve as a stand-in for some of the factors listed below, and the incorporation of social rules into the technology can make these rules harder to see and harder to change. In the examples above, much of the software is likely to change from the emergency room to the elementary school. The software that does not change (e.g. the operating system) may have been originally designed with one socio-technical system in mind (e.g. Unix was designed with an academic socio-technical system in mind). The re-use of this software in a different socio-technical system may cause problems of mismatch.
- *Physical surroundings.* Buildings also influence and embody social rules, and their design can affect the ways that a technology is used. The manager's office that is protected by a secretary's office is one example, and the large, cubical-filled office suite with no walls is another. The physical environment of the military supplier and the elementary school are likely to be quite different, and some security issues may be handled by this physical environment rather than by the technology. Again, moving a technology that assumes one physical environment into another one may cause mismatch problems.
- *People (Individuals, groups, roles (support, training, management, line personnel, engineer, etc.), agencies).* Note that we list here not just people (e.g. Ms. Jones) but roles (Ms. Jones, head of quality assurance), groups (Management staff in Quality Assurance) and agencies (The Department of Defense). In addition to her role as head of quality assurance, Ms. Jones may also have other roles (e.g. a teacher, a professional electrical engineer, etc.). The person in charge of the microcomputers in our example above may have very different roles in the different socio-technical systems, and these different roles will bring with them different responsibilities and ethical issues. Software and hardware designed assuming the kind of personnel support or expertise one would find in a university environment may not match well with an elementary school or emergency room environment.
- *Procedures (both official and actual) (Management models, Reporting relationships, Documentation requirements, Data flow, Rules & norms).* Procedures describe the way things are done in an organization (or at least the official line regarding how they ought to be done). Both the official rules and their actual implementation are important in understanding a socio-technical system. In addition, there are norms about how things are done that allow

organizations to work. These norms may not be specified (indeed, it might be counter-productive to explicitly describe how late to a meeting people of different status in an organization are allowed to be). But those who understand them know how to, for instance, make complaints, get a questionable part passed, and find answers to technical questions. Procedures are prime candidates to be encoded in software design. But to encode them, they need to be made explicit, at least to the designers, if not the users.

- *Laws and regulations.* These also are procedures like those above, but they carry special societal sanctions if the violators are caught. They might be laws regarding the protection of privacy, or regulations about the testing of chips in military use. These societal laws and regulations might be in conflict with internal procedures and rules. For instance, some companies have implicit expectations that employees will share (and probably copy) commercial software. Obviously these illegal expectations cannot be made explicit, but they can be made known.
- *Data and data structures.* What data are collected, how they are archived, to whom they are made available, and the formats in which they are stored, are all decisions that go into the design of a socio-technical system. Data archiving in an emergency room will be quite different from that in an insurance company, and will be subject to different ethical issues too.

Socio-Technical Systems change over time
An STS is configurable in all its elements, and this allows for change over time. By configurable, we mean that particular items in an STS can change over time, and that even among those items the configuration of one element may change. For instance, the particular mix of hardware and software within an elementary school's computing lab may change as the school gets access to the internet, or as more teachers begin to use the lab for their classes. But this change might also be reflected in changes in procedure (e.g. rules about access to certain web sites) and people (someone may need to take the role of censor in approving or disproving sites) and data (downloaded software, music, cookies, etc. on the machines' hard drives).

Change in an STS has a trajectory.
As the above example indicates, the change from a stand-alone computer lab to a lab connected to the internet may produce a coordinated set of changes within the socio-technical system. This coordinated series of changes in an STS is called a trajectory (ref to Kling). These changes can occur at the level of the STS itself, as in the internet example, or they can occur at the level of the individual parts of the system. For example, a different (but overlapping) socio-technical system supports the rapid evolution of microcomputers and their regular obsolescence. Elementary schools that do not keep up with this trajectory of the hardware in their system will find themselves quickly beset with problems.

These trajectories are most influenced by those with social power.
Since these trajectories are coordinated, who coordinates them? Research by psychologists, sociologists, and anthropologists in social informatics has led to the

conclusion that trajectories are most influenced by and usually support those with social power (refs). A few minutes reflection will make this statement seem self-evident, even nearly a definition of what it means to have social power. Social power is measured by money, influence, and other forces available to actors to help them influence change in a way that is in line with their goals. So, saying that trajectories are most influenced by those with social power is saying, in essence, that those with social power have power to influence trajectories. Not too surprising.

But the point is more than this. Trajectories usually support the status quo—those who already have power in a system. These are the individuals who get most influence in the construction of the technical specification of a technology, who pay for its implementation, and who guide its use so that it serves their purposes. And this influence may mean that the original purpose of software integrated into a social-technical system may be subverted by actors within that system. For example, Orlikowski (ref) documents how a technology designed to facilitate the cooperative exchange of information (Lotus Notes), was adopted differently in different parts of a large consulting firm. The organizational culture and rules of some departments (e.g technical support) were in line with the purpose of the software, and these departments adopted and even enhanced the product. Other departments (e.g. front-line consultants) needed to document that their time was spent in "billable hours" and saw no way to incorporate the Notes software into this scheme. These latter departments only used Notes in a perfunctory way, when they were required to, or adapted only those functions of the software that could be incorporated into the more competitive, "up-or-out" culture of their departments.

But there is still an ongoing debate among those who study such things about whether social power always wins in the contest to influence technological trajectories. There is, for instance, clear evidence that struggle groups, groups with much less political power than governments, have been able effectively to use computing technology (specifically the internet) to augment their power (refs). On the other hand, many repressive governments use technology in finely crafted ways to control the information their populations receive and to collect information on their people (refs).

Research on the use of technology in organizations has not supported earlier claims that the technology itself will produce a "leveling effect" in organizations. The idea, appealing at first, was that since technology enables easy communication among all levels of an organization, it will have the inevitable effect of "flattening out" the hierarchy in organizations that adopt it. By and large, this has not turned out to be true. Organizations can adopt computing technology with the intent of flattening their hierarchy, and it will help do this (ref). But organizations can adopt computing technology with the intent of strengthening the hierarchy (by, for example, installing keystroke level work monitoring on all computers) (ref). Again, it is the socio-technical system that produces the effects and structures the ethical problems, rather than the technology alone (refs).

Trajectories are not value neutral.
A moment's reflection should lead you to the conclusion that trajectories are rarely value-neutral. Trajectories have consequences and these consequences may be good or ill (or, more likely, good and ill) for any of the stakeholders in a socio-technical system. This is why ethical reflection should be a part of thinking about socio-technical systems.

The recursive nature of technology and social systems
The idea that recent social change in the "information age" is only the result of the revolutionary effects of technology is a form of technological determinism. This is the idea that technology "impacts" society. From the discussion here we can see that this is a simplistic view. Certainly computer technology influences how people act. This is what we mean by saying that technology is value laden in the way it structures human action. But it is equally true that when technology is actually adopted in a social system, it is adapted by that system. Technology influences human action in organizations and human action influences how the technology is designed and implemented. Technological determinism is a false and misleading simplicity.

The software designer or IT professional is one actor in this complex interaction. In the next section we will consider how being aware of the socio-technical system and its changing trajectory over time can help computer professionals do their jobs better.

Socio-technical systems and the practice of computing

Why should we use the language and approach of socio-technical systems in thinking about the design and implementation of computing systems? There are really two questions here. From the technical vantage point, we might ask what advantage socio-technical analysis provides in addition to standard software development methodologies. From the philosophical vantage point, we might ask what socio-technical analysis provides in addition to standard stakeholder analysis.

Socio-technical analysis vs. software engineering methods
Standard software development approaches certainly focus on hardware, software, and procedures and rules that should be incorporated into the software. To the extent that they concentrate on people and roles they are mostly interested in the explicit interaction a person has with the technology and on the place in the hierarchy the person occupies as a result of their role. The concentration here is most clearly on the visible and the documented. A socio-technical analysis, adds to this picture those aspects of work that are implicit, not documented, and based in informal political systems and actual (rather than ideal or documented) work practices. For the purpose of designing systems, a socio-technical analysis adds to standard concerns of efficiency concerns about skill development and social context. For the purpose of ethical analysis, a socio-technical analysis adds a concern for discovering the hidden practices and possible undocumented effects of a technological implementation.

Socio-technical analysis vs. "stakeholder" analysis
Standard stakeholder analysis in ethics spends most of its time looking (surprise) for stakeholders. This is really only one element of what we have identified as a complex

and constantly evolving socio-technical system. Procedures, physical surroundings, laws, data and data structures, etc. all interact to structure the particular ethical issues that will be relevant to a particular socio-technical system. Thus, a socio-technical analysis provides a more comprehensive and system-oriented approach to identifying ethical issues in a particular implementation of a technology.

Practical Implications
So what does a socio-technical approach to computing mean for the computing professional? First, it helps to identify the kinds of power (both intentional and unintentional) that computer professionals have when they design and implement systems (ref to Huff unintentional power paper). It also helps to identify the countervailing power that other actors in the system have (power users, clients, consultants, industry reporters, regulators etc., see the text box on overlapping systems). And it shows how arguments over what seem to be simply technical aspects of efficient system operation can really be a part of complex power struggles.

Awareness of these issues certainly makes life more complex for the computer professional. But an awareness of organizational complexity is an important part of even standard software design methodologies (ref to Man-Month) and is crucial to good system design practices.

In addition, awareness of these issues also makes it possible to adopt practices that lead to a more socially and ethical practice of system design. Next, we will consider some specific approaches toward this end.

Text Box: Are Computers Value-Neutral?
The claim that computer technology itself cannot "have" values seems on its face a compelling one, and yet we argue in this book that it does. Let's give the "value neutral argument its proper due and see whether a case can be made on its behalf. There appear to be five approaches to this argument and we will examine each one. In the end, we will conclude that the claim that technology is value-laden does need to be modified, but not so much as to relieve computer professionals of any responsibility for the technology they design.

Only people have values. This argument is the relatively straightforward claim that inanimate objects cannot have values. A piece of software sitting on a hard drive may be said to have logic and structure, but the claim that it has values is to talk about it as though it were human. Only humans (and perhaps some other animals) can have values.

Reply: For the sake of argument, let's agree with this premise, that only people can "have" values. But it is certainly true that the values (like efficiency, or concern for privacy) that the designer had when designing the software can affect the software's design. And it is true that the design of the software can shape the actions of the user by allowing certain actions and forbidding others. In this way, the users' actions can be made to conform to the values on which the design was based. So, even if we accept that the software itself cannot "have" values, it is still true that the values on which the design is based will be expressed when the software is actually used by someone (to do things like control a medical device or match personal data). So, the more careful expression of the "software is value laden" claim should be stated as "software-in-use will shape the use so that the value on which the design is based will be expressed in the action." This is a bit long winded, but has all the appropriate hedges in it.

The computer is a general-purpose tool. This argument it that since the computer is a general-purpose tool, it cannot be said to be value laden, because it can be programmed to an infinite variety of tasks. Thus the nature of the device is not constrained by values that prefer one task over the other.

Reply: This argument rests on a sleight-of-hand by what we mean by "the computer." Turing machines, appropriately instrumented, are general-purpose devices. But as soon as some specific task is programmed into the machine, then the design of that program constrains what the machine can do. So, yes, it *may* be that a real general-purpose device, like a Turing machine, is "value-free" (though we may not be willing to concede even this). But as soon as any program modifies the general-purpose nature of the machine, it loses its "value-free" claim. So, any computer that is actually programmed to do anything, can be said to have values from the design of the task programmed into it.

Reuse shows technology cannot constrain action. This argument accepts the idea that particular computer programs in use might be value laden. But it makes the entirely reasonable observation that programs and pieces of programs are often reused for purposes beyond the wildest dreams of their designers. The original designers of the internet (ref) and of the web (ref) had no idea how widely varying a use their designs would obtain. So, if we cannot predict how software or hardware will be reused, then we cannot really say that it constrains the actions of its users.

Reply: This argument makes a profound point: technological determinism is too simplistic an approach, and we cannot assume that the values that inform a program will always inform that program's reuse. But it generalizes too far from that point. We must assume that the design of a program constrains its users in some sense, otherwise, it would be a completely general-purpose machine (and not much use for any specific purpose until it was programmed for it). So, the argument makes an important claim, but goes too far.

Adoption and use by people is what implements values. This is the "guns don't kill people, people use guns to kill people" argument. There is a great deal of force to this argument. Software or hardware cannot on its own, without being set into motion at some time, take any action. And so it cannot "implement values" without there being a tool-user who uses the tool for that purpose.

Reply: Again, this is a profound point, and one that bears repeating. It is software-in-use that implements values. But this claim again goes too far by absolving the tool (or more appropriately, the tool's designer) of any responsibility for the shape the action takes. Some guns are designed to kill people. Some are not. Sponge cakes are rarely designed to kill people (though with appropriate creativity, you could think of ways to use a sponge cake for this nefarious end). Again, design can make a tool easier to use for some purposes than others, and thus can constrain or encourage certain activities.

The designer is a simple agent of the client and so not responsible for values. This is not really an argument that technology cannot implement values. Instead, it is more an attempt to distance the designer from responsibility for the design. We will talk much more about these "moral distancing strategies" in chapter 4. Still, if the designer is only doing the bidding of the client, then it is the client who is responsible for any values that structure the design.

Reply: In the end, this argument does not work, unless you are willing to say that you would design technology for any end whatsoever, including genocide. If there are any programs you would not design or build because of their effects on others (or any designs you would not use because their effects on others were unacceptable), then we are now playing the game of deciding what the grounds are for placing a program in this proscribed status. And this is a game of deciding what responsibilities *you will take as a computer professional*, rather than disclaiming all responsibility.

Ethics Into Design

Whenever you are involved in designing or implementing a computing system, you at least hope the system will have the immediate effects for which it is designed (e.g. efficient data processing, safe control of a machine, etc.). But Huff (unintentional power ref) argues that systems also have unintentional effects, effects that the designers did not anticipate. Most of the cases we have included in this book involve the effects of systems that were not initially foreseen by the designers.

Standard project management approaches for software development often attempt to address these issues (refs), but they are limited in doing so because of their relatively narrow focus on efficiency and cost effectiveness (refs). The problem often lies in the area of requirements analysis, where a narrow focus on efficiency can overlook important element of the socio-technical system that will support or hinder the eventual successful implementation of the system.

Only procedures that include asking larger questions about the appropriateness of the requirements themselves can really address this narrowness of focus. We saw in this chapter that computing systems are embedded within larger and more complex socio-technical systems. All the methods we present here are designed in some sense to influence the design of the software based on information about the socio-technical system.

What all the methods have in common is making sure that early design decisions are made based upon a consideration of larger contextual issues rather than narrowly focused technical and economic ones. Some of the methods have been confined to specific areas of application, while others claim a more general application. We will start our review with some of the more narrowly applied methods, and then move to the general ones. In chapter 2, we walk step-by-step through one of the approaches in detail as a template for how concern with ethical issues can inform a system design.

Socio-Technical Design Methods for Specific Contexts

Both of the methods reviewed in this section have been tailored for specific environments. In one case (CSCW) the tailoring is for software applications of a particular type: those that support computer supported cooperative work. The second method (Participatory Design), was originally tailored for a specific environment, the Scandinavian context in which cooperation between strong labor unions and management, with government support, is a long tradition. In the end, both methods could be generalized to other environments with only a little effort, and some attempts have been made to do so with Participatory Design (refs) and with CSCW methods (Kling Informatics refs).

CSCW

Computer supported cooperative work is an area in which systems are designed to support the work of groups. The work in this area is concentrated on those values that are most relevant in CSCW systems, for instance, privacy (Hudson & Smith, 1996) and trust (Rocco, 1998; Yakel, In press). Some analysts have observed that often values like cooperation have been assumed by CSCW designers only to find that in actual implementation competitive values in the organization make the software unlikely to be successfully adopted (Orlikowski; ref to hospital study, ref to Jon Grudin).

Nevertheless, the approach involves the iterative design of a product with evaluation based on social science methods at each iteration. The methods include things like participant observation, rapid prototyping, interviewing, unobtrusive observation of the product in use, focus groups, questionnaires, and collection of archival data. (See the in-depth section for more information on these methods). Usually, these evaluations are done by design team members who are specialists in social science methods (refs). Often the design teams involve collaboration between grant-funded social science research groups and either industry or grant-funded software development experts (ref to examples).

CSCW work is gaining momentum and is now strong in both academia and industry. The work of those doing CSCW based design is regularly documented in conferences held in both Europe and North America (refs). The methods of socio-technical design originally pioneered in CSCW applications are now being more broadly applied in a area called *Informatics*, a term that implies the design and use of information technology in a way that is sensitive to the socio-technical context (refs). This approach is more popular in the European community than in North America, though there are several programs in the US that have adopted the approach (refs). As the work in Informatics has expanded, it has become more of a general purpose approach to socio-technical design and less tied to CSCW contexts.

Participatory Design
Participatory design is an approach that attempts to integrate democratic values into system design (Bjerknes, Ehn, & Kyng, 1987; Greenbaum & Kyng 1991; Kyng & Mathiassen, 1997) by involving potential users of the system in intensive consultation during an iterative design process. The approach evolved in Scandinavia (particularly Norway, Sweden, and Denmark) as a collaborative effort between unions and technologists. The unions wanted to use technology to enhance the jobs and skills of their members without contributing to unemployment. The technologists were interested in designing technology that would enhance the democratic values (e.g. empowerment of workers) that the unions supported. The 1981-1986 UTOPIA project (ref) was the first large-scale attempt to do this.

Participatory design approaches incorporate the users into the design process as significant partners. This is easier when the clients for the software product are in fact the users (as represented by the unions) than it is when the client is the employer. The methods used include:

- Early, intensive interaction among the users, the software designers, and the associated social science researchers. In these interactions, the users are treated as the experts on how the work is done and are recruited to help envision the shape of the new product.

- Requirement specifications based on these early interactions, and modified by early mock-ups of the design using low tech approaches (e.g. wooden mice, waxed cards).

- Iterative design based on the requirements. The iterative process always includes the users on an equal footing as the designers rather than relegating users to the position of test subjects in field trials.

Participatory design has been encouraged by many American academics (refs), but it has been slow to catch on, partly because of the intensive time commitments involved and partly because the specific democratic values that are inherent in the process are perceived to be in conflict with the hierarchical workplace. But Bødker et al(2000) make the point that the democratic values associated with participatory design really have a double life. On the one hand, they are rooted in the Scandinavian radical political tradition that may not translate well to other cultures. On the other hand, participatory design involves designers in significant learning about the perspectives of the user. It is this intimacy with the world of the user that contributes to the success of products using participatory design. And the methods of participatory design may be the best way to achieve this intimate knowledge of the users and of the product-in-use. Bodker et al (2000) recognize that this means the method will become increasingly detached from its original commitment to radical democratic values as mainstream software developers adapt it to their use.

An interesting project in Sweden called the User Award has developed in this tradition. This award, given by LO (the Swedish blue-collar union confederation), is regularly given to software products that give good support both technically and socially in their implementations (ref). It uses many of the principles from Participatory design in its evaluation of software for the award.

General-Purpose Socio-Technical Design Methods
We review here four methods that claim a more general applicability to software design. In principle, each of these methods could be used in any software development project to allow the designers to adapt their design to the socio-technical system for which the product is intended. The originators of these methods claim they not only help to ensure that the software addresses important ethical issues, but also that the method helps to ensure a product that is better fitted to its eventual implementation, and thus more likely to be successful. All the approaches have a focus on the ethical issues that surround the design and implementation of a system and assume that addressing these is in fact the way to shape the software so that it best fits with the socio-technical environment.

The ETHICS method
Mumford's (1996) work was inspired by her early association with the Tavistock Institute in London, a center founded in 1946 for the study of the interplay between technical systems and human welfare (Trist, 1981). Mumford is thus one of the earliest pioneers in socio-technical system design. The method's application to other technical systems has been reasonably successful (Brown, 1967) and has seen a few successful implementations in software systems (Mumford & MacDonald 1989; Mumford, 1996). The accompanying table provides an overview of the process. Mumford (1996) provides much additional detail along with examples of systems that have used the process successfully.

Steps in Mumford's (1996) Socio-Technical Design Process
1. Diagnosing user needs and problems, focusing on both short and long term efficiency and job satisfaction. 2. Setting efficiency and job satisfaction objectives 3. Developing a number of alternative designs and matching them against the objectives. 4. Choosing the strategy which best achieves both sets of objectives. 5. Choosing hardware and software and designing the system in detail. 6. Implementing the new system 7. Evaluating the new system once it is operational.

The work of the Scandinavian Participatory Design groups was influenced by Mumford's early work at Tavistock. Mumford's approach and Participatory Design share some features. Both are focused on quality of job life for users of the software and thus both methods are primarily designed for workplace systems. Both are iterative and begin with early interaction with the users. Mumford's emphasis on developing several alternative designs can be interpreted as a difference in approach, but rapid prototyping in Participatory Design also involves the use of different approaches in a testbed environment.

Social Impact Statements

Shneiderman's original (1990) proposal was based on an analogy with the government requirement in the USA that construction projects complete an Environmental Impact Statement to determine the project's effect on the surrounding environment. He proposed a Social Impact Statement be incorporated into the software development process of projects that were large enough to warrant it. This statement would begin by listing the stakeholders (people and groups likely to be effected by the software), would attempt to determine the systems effect on those stakeholders, and then propose modifications in the requirements or design of the system to reduce negative and enhance positive effects. Shneiderman and Rose (1996) is an attempt to implement this approach in a system. Huff (1996) has proposed the integration of particular social science methods into the Social Impact Statement as a way to increase its sensitivity to the social context in which a system is designed. It will be this integration of social science methods into an SIS that we will explore in depth later in the chapter.

Software Development Impact Statement

Gotterbarn and Rogerson (2002) provide a method and supporting software to investigate the social issues associated with the implementation of a system. This approach is as simple as asking what the effects of each of the system's tasks will be on each of the stakeholders who are relevant to that task. But this simplicity is illusory because of the combinatoric explosion that occurs when you cross all tasks with all stakeholders and ask a series of questions about each of these combinations. The SoDIS (Software Development Impact Statement) software that Gotterbarn and Rogerson (2002) present helps to control this complexity. The steps involved in SoDIS are:

Steps in completing a Software Development Impact Statement (SoDIS)
1. Identify Immediate & Extended Stakeholders
2. Identify requirements, tasks or work breakdown packages
3. For every stakeholder related to every task, record potential issues
4. Record the details and solutions to help modify the development plan

The next table provides a sample of the questions that are to be asked for each task in the software development project and for each stakeholder. The person doing the SoDIS analysis asks each question for each task and each shareholder and makes a note of concerns that might arise. Some concern may need additional investigation to allow a decision to be made about them. After the SoDIS is completed, the result is a comprehensive set of recommendations for the software development project with specific proposals for changes in the development plan targeted at identified problems.

Sample questions from the Software Development Impact Statement For each question, ask does task I, for Stakeholder J
Cause Harm • Cause loss of information, loss of property, property damage, or unwanted environmental impacts • Cause the unnecessary expenditure of resources

- Cause risks of which the stakeholder is unaware
- Violate the privacy or confidentiality of the stakeholder
- Risk injury, loss of life, pain, or disability
- Expose stakeholder's data or confidential information to unauthorized access or alteration?

Cause Pain

- Keep hidden any possible dangers to the user or public
- Involve the design of approval of software which may cause a risk of injury or loss of life
- Lessen the quality of the stakeholders working life
- Lead to software that may not be safe

Limit freedom

- Limit access for disabled users
- Fail to consider the interests of employer, client, and general public
- Ignore some of the stakeholders needs

Conflict with your duty

- Neglect quality assurance processes and review by the stakeholder
- Avoid full acceptance of responsibility by the stakeholder
- Cause ineffectivenees and inefficiency as perceived by the stakeholder
- Involve a conflict of interest with the stakeholder

SoDIS analysis, therefore, is primarily a prospective method done at the beginning of a project, and is not done iteratively throughout the lifecycle. It allows developers to look ahead in a comprehensive manner at the issues that a project might raise, and to anticipate those issues at the beginning of the project.

Value-Sensitive Design

Friedman, Howe, & Felton (2002) report on the use of a design process called Value Sensitive Design that iterates among conceptual, technical, and empirical tasks throughout the software design project in an attempt to account for important human values that are affected by the system. The idea is to start with philosophically informed analysis of relevant values, then identify how existing and potential technical designs might enhance those values, and then use social science methods to investigate how those values are affected in various stakeholders related to the system.

An example of this approach in use is Friedman, Howe, & Felton's (2002) redesign of the cookie handling facility in the Mozilla web browser. Cookies are bit of information that are stored on the user's machine by a particular website. They are often used to allow sites to customize their interaction with the user, but are also used to track the user's behavior (sometimes even across many sites). Most browsers have a facility that allows the user to modify the way their browser handles cookies (e.g. not accepting any, only accepting some, exploring the contents of the cookie). Friedman et al (2002) wanted to redesign the Mozilla browser's cookie handling facility to maximize *informed consent* by the user. They first engaged in philosophically informed analysis of the idea of informed consent, and came to the conclusion that informed consent to an action (like allowing a cookie to be stored on your machine) involved the following five characteristics:

1. Disclosure of accurate evidence about the harms and benefits to be expected
2. Comprehension and accurate interpretation of the disclosed information
3. Voluntariness of the action (it is not controlled or coerced)
4. Competence to give consent (having the mental, emotional, and physical ability to give consent)
5. Agreement having a reasonably clear opportunity to accept or decline

They then began a retrospective analysis of how browsers accomplished all these aspects of the task of giving informed consent to cookies. Based on this analysis, they proposed design goals for their redesign of the Mozilla browser cookie facility. For instance, one goal was to "enhance users' local understanding of discrete cookie events as the events occur." They then proceeded through several prototypes (in early stages using "thought experiments," then graduating to mock-ups and then actual software protoypes) while using social science methods to evaluate their success with each prototype. During this process, they discovered that they needed an additional characteristic of informed consent:

6. Minimal distraction from the user's primary task (the process of consent should not overwhelm the user).

Their final product, the Cookie Watcher, allowed users in real time to see the effects of cookies on their machine without too much distraction from their primary task. If they decided to pay more attention to a particular cookie event, they could then turn to this task.

Like most other approaches, this methodology is a new development and though it has shown interesting successes (Freedman, Howe, & Felton, 2002; Friedman & Kahn, In press) it still awaits more careful validation. Though it is a general approach that allows the developer to concentrate on any particular value or set of values to guide the design, it is different in scope from the SIS and SoDIS approaches. Those approaches are broad brush attempts to discover and react to the social and ethical issues or values that are relevant to any particular design problem, while value-sensitive design chooses the values of importance and attempts uses them to guide the development process.

Summary of Socio-Technical Design Approach

All of these approaches differ in some respects. Some are mostly prospective, attempting to predict the issues that a particular implementation will face. Others are iteratively incorporated in the design process. Some are focused on maximizing particular values (trust, privacy, democracy, informed consent) while others are designed to make the developer aware of the range of social and ethical issues that a project presents.

But all are based in the assumption that gaining knowledge about the socio-technical system in which a technology is embedded will help the designer to design software that at least avoids some ethical trouble spots, and perhaps helps to increase the influence of particular values. Most of them of them require that social science methods be used to investigate the socio-technical system. SoDIS does not require the use of data collection, but it certainly allows for it to be integrated in answering any of the questions.

There have yet to be definitive empirical trials of these methods that would establish them as clear improvements on the standard software development process. Then again, definitive empirical trials in evaluating software development methods are hard to come by (ref). But there is rich case-based evidence that these methods help in the design of software that is more ethically sensitive and is better fit to the socio-technical systems in which the technology will be employed.