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 1
Introduction

Version 1: 01/30/05 by Chuck Huff
Version 2: 02/03/05 by Chuck Huff
Version 2.1: 02/09/05 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

The time is right to get serious about this. As software becomes increasingly dominant in the IT industry, and, indeed, in everything else, there is an obvious need for a professional-level recognition. Far too much is placed on particular credentials for specific products or applications without regard to the bigger picture. The result is poorly engineered software projects.

the IEEE-CS/ACM Joint Task Force on
Software Engineering Ethics and
Professional Practices (ref, 1999)

This book is about this confluence of the two meanings of *good* in its title: Good computing as in *computing done well*, and also as in *ethical computing*. The statement above begins the web page that displays the IEEE/ACM code of ethics in Software Engineering. The odd thing about this statement is its last sentence. Concern for ethical practice is framed here not as "rules for being a good person" but as being integral to *well engineered software projects*. We agree with the authors of the Software Engineering Ethics code that good software engineers need to know enough about the "bigger picture" to do software engineering that works well. By the time you are done with the text, we hope you will agree.

Some of the cases you will read in this text will, we hope, lead you to agree with the Software Engineering Ethics Task Force. The radiation treatment device called Therac-25 killed several people and injured many others, in part because of flaws in the software design. The damage that may have occurred from faulty analogue-to-digital converters in the Hughes Aircraft case is unknown, but these chips were used in numerous military applications, including fighter aircraft and guided missiles. The privacy invasion and death threats that occurred at the University of California, Irvine, in the Machado case were enabled by operating system choices in Unix. The significant financial risks that Texas faced when implementing computer-intensive pedagogy in its public schools was structured by the way computing was approached in that case. The clear financial losses that the medical technology company Biomatrix incurred came about because of the way internet financial discussions on Yahoo were structured by the software.

After you review these and other cases, you are likely to be more aware that issues like these are intimately intertwined with how software is designed, constructed, implemented, used, and supported. But this text is not primarily about convincing you that you ought to care about the ethical (and un-ethical) effects of software you design. We expect you already do, and that you will learn some things that will help you care more.

But more to the point, we hope you will learn some things that will help you to care more effectively. We will be presenting a set of skills and knowledge you can learn and practice. A set of skills and knowledge that, when combined with your expertise as a computer scientist or software engineer will allow you anticipate as least some of the problems that might arise when software is actually put into use. And not only to foresee these problems, but to construct solutions that resolve them. The outcome will be, we hope, as the Software Engineering Task Force on Ethics claims -- you will be able to design, construct, implement, and support better software.

We take our lead in this from the convergence of four trends we have been following for several decades: philosophical work on virtue ethics, the psychology of moral practice, social science research on technology, and developments in software design processes. The book presents work you will do in projects and case discussions to practice the skills and habits of good computing design. While doing so you will apply basic and intermediate ethical concepts and adapt procedures we provide to fit the complexity of the real-world situations of the projects and cases with which you will be faced. The approach is essentially computing ethics as a laboratory class in which the projects and cases provide the laboratory.

From the moral psychology literature we take the idea that moral judgment and action in a domain (e.g. software engineering) can be learned by intense practice in a setting that provides immediate feedback, much like coaching an athlete or other expert (see Huff & Frey ref for a review). The practice needs to be based in basic knowledge about concepts like "justice" and "duty" but also in intermediate concepts like "system safety" "privacy" and "intellectual property." Most of the basic concepts will come from Sections I and III of this book, the framework and theory sections. Most of the intermediate concepts will come from Section II where detailed cases will be used to explore concepts like safety, privacy, and intellectual property. Two of the primary skills you will practice are *moral imagination*, the ability to put oneself in another's position and *moral creativity*, the ability to construct solutions to moral challenges.

From philosophy, we take a close reading of Aristotle and the subsequent revival of the Virtue Ethics literature to understand virtues in computing as well-practiced skills, based in knowledge associated with a craft, and learnable by practice and coaching. The primary turn that virtue ethics have taken in philosophy today is to move us away from asking "What is the best rule for deciding if an action is ethical?" to asking "What virtues should I practice to best achieve my potential?" In Chapter 12 you will find an introduction to virtue theory, along with overviews of other basic ethical approaches. In several other places throughout the book you will find some reflection on particular virtues (e.g. reasonableness) that are specifically applicable to the context of software engineering.

From social science research on technology we take the idea that you cannot understand a technology in isolation. Instead you must understand what is called the socio-technical system, including the software, hardware, people, roles, rules, institutions, procedures, and data that together produce consequences that one might call ethical or unethical. We also take some methodological sophistication about how to study these socio-technical systems in a way that might inform the design of software and hardware. You will find reflections on the concept of socio-technical systems in chapter 11, and a presentation of methods in Chapter 2.

From software design we take the idea one can integrate concern for the larger socio-technical system into the process of the design of software. In Chapter 11 we present some specific methodologies that attempt to do this, and in Chapters 2 through 5 we present a method that can be used to integrate an understanding of the socio-technical system into software design. In the rest of this chapter, we introduce this approach.

What you won't get in this book

This book will be relentlessly practical in its approach. We provide overviews of the theory in various areas only to the extent that they are required to understand the basic and intermediate concepts and practice the skills.

This means you will not get much basic theory in philosophical ethics or in ethics as it applies to large policy issues in computing. Thus, we cover theoretical and policy issues in intellectual property only to the extent that they help us understand how the systems we design and use interact with these issues.

There are fascinating philosophical issues in the area of computer ethics. Where does the concept of property come from? Can it be justified in the face of huge disparities of resources in the world? How does the transmutable nature of software make it difficult to understand it as property that one might sell? How should we balance the needs of society for useful information with the property rights of information owners?

Privacy is another difficult philosophical puzzle. What is the vaunted "right to privacy?" How can one defend it? What information about oneself is necessarily public? What obligations does government have to encourage privacy? Are there different kinds of privacy (e.g. accessibility, informational)?

Some of these issues these are covered in our cases, but there are other philosophical issues we do not approach at all. For instance, are there any decisions that computers should not be allowed to make, no matter how accurate they are? Is there a point at which we might want to ascribe moral responsibility to artificially intelligent machines? Or even a meta-ethical question such as whether computing technology poses fundamentally new questions for ethics.

We suggest you supplement your use of this book with another text that covers these issues in some depth. Deborah Johnson's classic and much revised *Computer Ethics* (ref) is one good choice as is Herman Tavani's *Ethics and Technology* (ref). For a good overview of Virtue Ethics, we recommend (ref). There is no recent good overview of moral psychology, but we recommend looking at (Lapsley ref) and (ref). For Software Engineering, we recommend (Kazman ref) and for the integration of ethics into software engineering (Friedman ref) and (ref).

An iterative model of ethical problem solving

Good practice in computing in the both the technical and ethical senses follows from a careful and methodological approach to the problems that arise in practicing this profession. We propose an iterative model of ethical problem solving that is designed to integrate with various approaches to software development. The stages we propose will look familiar to you: *problem specification, solution generation, solution testing, and solution implementation*. They are similar to many linear software development approaches, and we think this similarity will make this process easier to integrate it into software development. First, we will overview the individual stages and then look at each stage in some depth. Finally we will ask how these stages can be incorporated with three different approaches to software development process.

1. *Problem specification:* Here, you will learn to specify the socio-technical system that influences the software in question. Part of this stage involves recognition of values that are inherent in the system, and of the conflicts that may arise between particular values, say, between efficiency and safety.
2. *Solution generation:* In this stage*,* you will try to resolve identifiable value conflicts in a way that is feasible in the socio-technical system. Sometimes this involves changes in software design or requirements, but often it involves recommending change to other aspects of the socio-technical system. A characteristic procedure of this stage is brainstorming.
3. *Solution Testing*: The solutions developed in the second stage must be tested in various ways. We introduce procedures for testing the worth of proposed solutions that use philosophical approaches and psychological criteria.
4. *Solution implementation*: The chosen solution must be examined in terms of how well it responds to various situational constraints that could impede its implementation. What will be its costs? Can it be implemented within necessary time constraints? Does it honor recognized technical limitations or does it require pushing these back through innovation and discovery? Does it comply with legal and regulatory requirements? Finally, how does it respond to the general social and political conditions surrounding implementation?

Carolyn Whitbeck (ref) has already recognized the similarity between problem solving in ethics and engineering design. We build on her work by adapting the software development cycle to ethical problem solving in computing. By the end of this chapter, we will see that good computing involves a synthesis and harmonization of both the technical and ethical senses of good. One cannot be a good technical computing practitioner without at the same being a good ethical computing practitioner.

Problem Specification

Computing technology and practice does not take place in a vacuum; it always occurs within a socio-technical system which affects it and which it affects. Moreover, computing technology and the socio-technical system in which it is embedded are never value free. The first step in problem specification is to determine what the social technical system under evaluation looks like. A socio-technical system (STS) is the social context within which computing technologies function and computing practice takes place. We can begin to understand the STS by describing its constituent parts:

- *Hardware and software.* Social-technical systems have computer hardware and software. A classroom in a public school has a desktop computer that the teacher uses for presenting material to the students. This is the hardware. The teacher gives these presentations in PowerPoint and prepares handouts for students in Word. These programs constitute software. A doctor's office has a computer (hardware) on which is installed a diagnostic program (software). But software can also be integrated into appliances and tools in a way that make it difficult to distinguish.
- *People, Groups, & Roles.* Computing technologies are used by individuals who carry out tasks that are required by social, job-related, or professional roles. For example, teachers use computers to help them take attendance, record and calculate grades, and prepare classroom handouts. Taking attendance, recording grades, and preparing handouts are tasks required by the role of teaching. People and the actions they perform are structured and related by these roles. Corporate organizations, in turn, relate and coordinate social, job, and professional roles.
- *Procedures.* STSs are filled with different sorts of procedures that people carry out within different organizational, professional, and social contexts. For example, paper making at a pulp mill requires carrying out a certain procedure where wood is converted into pulp, which is converted into paper. Universities require professors to carry out certain procedures to request and liquidate travel funds. In the Therac-25 case, procedures for setting up the machine were part of what caused the deadly accidents.
- *Laws and Regulations.* Social-technical systems are informed by laws and regulations. A computer chip manufacturing process that produces toxic byproducts must respond to environmental regulations governing the disposal of such byproducts. Investors who lose money following the advice of a computer driven investment program might sue the program's designers to recover their losses. In the Machado case, the FBI uses the case to establish a legal precedent for classifying certain forms of flaming as hate mail.
- *Data and Data Structures.* Finally, a socio-technical system description should include an account of the data and data structures manipulated and stored by the computing system. The data about the identity of Yahoo users played an important part in the Biomatrix case. How that data was stored, its confidentiality, and the circumstances under which this confidentiality would be violated need to be understood to make sense of crucial events in this case.

In chapter two you will learn methods you can use to specify the relevant socio-technical system for a case you are trying to understand or for an unfamiliar existing or proposed new system.

Problems arise in three ways from a socio-technical system.

- From a mismatch in values embedded in the computing system and in the larger socio-technical system.

- From a mismatch in values in the larger socio-technical system that the computing system exacerbates.
- From overlooked outcomes of the computing system that violate values in the larger socio-technical system.

An important point here is that computing technology is value laden, that is, computing technologies (CT) exhibit structures that support certain values (see chapter 11). Second, computing technology is always embedded in a socio-technical system (STS) that also has values. Let's look at each of the three ways in some more detail.

CT and STS value mismatch
A CT may be designed to support particular values (e.g. cost efficiency). One reason for the particular Therac-25 medical radiation therapy machine design was that it decreased manufacture cost (and thus served other values, like making the therapy more widely available). But this design introduced complexity into the machine that was a root cause of the safety problems. So in this case, a value designed into the CT (cost effectiveness) conflicted with a value that the larger socio-technical system had (safety). In the Machado case, a program that promoted connections among users (finger) was used to harass and threaten users. Again, a mismatch between CT values and those of the STS. It is only by looking closely at the values embedded in the computing technology that these sorts of issues can be predicted and avoided.

Internal STS value conflict exacerbated by technology
Of course, one might say that the value of cost effectiveness was really one already embedded in the socio-technical system. But a clearer example of this is in the Biomatrix case, in which financial bulletin boards were fraudulently used to spike the value of a company's stock. In this case, there were people in the socio-technical system with competing goals, and they used a technology, the bulletin board, that exacerbated their conflict. Often these are difficult to predict, but again, analysis of the socio-technical system can help.

Overlooked Outcomes
Often the effects of introducing a CT into a system are remote and outside the control of the designer of any specific system. But there are also more direct social effects of design decisions in particular systems, and these decisions are ones over which computer professionals may have influence. The distinction between remote and direct effects of computing is helpful, but not clear cut. It is crosscut by the intentional effects of the computing system (efficiency, unemployment) and its unintentional effects (deskilling, component reuse in other applications). Even basic research in computing like the development of systems of fuzzy logic can have intentional effects (making medical diagnosis more reliable) and unintentional ones (making computer database matching more efficient). So, combining these classifications of the effects of computing may give us a better view of the issues at stake.

Some examples of different kinds of social impacts of computing

	Basic Research (example: development of fuzzy logic systems)		System Development (example: data monitoring in banking call center)	
	Remote	Direct	Remote	Direct
Intentional	Increased reliability of medical diagnosis AI systems	Allows categorization algorithms in difficult areas	Better integration of customer data across sales and support	Increased productivity in call center
Unintentional	Increased privacy violation by marketers using data matching techniques	Revisions of conceptions of human rationality	Increase in job turnover among workers	Decrease in job satisfaction among call center workers

For simplicity sake the dimensions are shown as categories, but each is obviously a continuum. Research is basic to the extent that it is solving intellectual *vs.* practical problems, though clearly many actual projects are some mixture. Also, an effect is direct when its outcome depends little upon the presence or absence of other factors (e.g. the presence of a union, or regulations, or reuse of the system for a different purpose). An effect is intentional to the extent that the developers and client foresee and desire that outcome and design the system to produce it.

What this categorization makes clear is that there are social effects of computing that are more under the control of software engineers and there are those over which designers have little control (or at least share control with many other actors). Again, this is a continuum rather than a dichotomy. Creative designers can often find technical solutions to address even remote effects of system design, once they are aware of them. And the point of problem specification is to become aware of them.

Solution Generation

Not much emphasis has been placed on solution generation in ethics. Instead, ethicists concentrate on moral judgment and moral reasoning: given the availability of a set of alternatives that respond to a situation, which is best from the standpoint of moral judgment. But where do these alternatives come from in the first place? In this section, we look at ways in which moral considerations guide the process of designing solutions to the problems that emerge in the problem specification stage. We also look at guidelines for integrating ethical considerations into solutions to general problems in computing.

Consider the following scenario:

A coop student notices that workers who use catalyst A in a manufacturing process habitually violate safety protocols. She finds this disturbing since she knows that A causes cancer. During a meeting of her work group, she expresses her concerns and recommends switching to catalyst B which is non-carcinogenic. The leader of the group, a senior engineer, tells her to drop the issue. Because the workers are violating safety procedures, ***they*** *are responsible for whatever harm may occur, not the company. Furthermore, he has already looked into catalyst B, and it is much more expensive than A. What should the coop student do?*

Problem specification would lead us to look at the way safety is embedded in the STS of this case. Examining the *physical surroundings* to the manufacturing process (including worker exposure to catalyst A and its carcinogenic properties), *the people and groups* (the workers, their supervisors, the company), *the safety procedures* (which are being violated for unknown reasons), and *the legal context* (potential lawsuits brought by workers who are harmed by exposure to A), we specify that the fundamental problem is a conflict between safety values and financial values. If we give priority to financial values, then we stay with catalyst A since it is the cheaper of the two. However, if we choose to respond to emerging safety/health risks, then we should seriously consider switching to B. In the solution generation stage, we need to work on designing a solution that responds to this value conflict.

The first thing to do—should time and cost permit it—is to gather more information. Is catalyst A really carcinogenic? Is B clearly preferable from a health/safety standpoint? What are the safety procedures for handling A, and why are workers violating them? Do workers know about the carcinogenic properties of A and fully understand the risks of exposure? Why is B more expensive? Could the manufacturing process in which the catalyst is being used be redesigned to better accommodate B? Perhaps the conflict between safety and financial values and the disagreement of the coop student with her group leader merely stem from a lack of information and could be readily solved by collecting good, solid information.

If gathering information does not dissolve the problem or is not possible, what other options are available to the coop student? Anthony Westin (ref) recommends employing the "intermediate impossible" strategy. Here we imagine what would be the perfect solution and then make the minimal changes necessary to make this perfect solution feasible. A perfect solution would integrate the conflicting values of safety/health and cost. For example, suppose a computer control system could be developed (quickly and cheaply) that would direct the process in which catalyst A is used that would allow workers to monitor the process from a safe, protected environment. This solution would deal with the safety/health value because workers would no longer be exposed to catalyst A. On the other hand, it would also deal with the financial value since A, the cheaper of the two catalysts, would be retained. In its form, this solution solves the problem by integrating the two conflicting values. However, problems might arise when we turn to implementation that would require modifying the solution somewhat. (How much would it cost develop the computer control system? How expensive would it be to purchase the required equipment? Is there time to implement this solution? And so forth…)

We now have two general strategies for designing solutions that integrate ethical value into the solution: gathering information and designing a solution that integrates the conflicting values. What if these two options are not available? Failing to find a solution that integrates the conflicting values, perhaps we can design a value compromising solution. For example, suppose the coop student finds out that productivity pressures have led workers to abandon the time-consuming safety procedures? In this case she might propose keeping catalyst A but informing the workers of the health risks of exposure to A, refining the safety procedures to make them less time-consuming, and relieving some of the productivity pressure. In this case the company might give up something to safety, namely, it might have to concede more time to the manufacturing process. On the other hand, it would gain some of the financial value it enjoys by retaining A, the cheaper of the two catalysts. On the other hand, the workers would give something up (they would have to conform to safety procedures) but they would reduce risks produced by exposure to A. The conflicting values have not been fully integrated in this solution. Nevertheless, the value compromise allows for a partial realization of each value.

But suppose that it becomes impossible to realize even a value compromise. In this case, the best option remaining would be to rank the conflicting values and choose to realize the more important of the two while setting the other aside. In this case, safety/health has priority over financial considerations. Hence, the company should adopt catalyst B, even though it is more costly than A. Companies have an ethical obligation to honor the safety of their workers and should sacrifice other values to bring this about.

Finally, return to the group leader, the one who told the coop student to drop the issue. Suppose he refuses to listen to any more arguments, even to good arguments. He bullies the other members of the group and threatens the coop student with a devastating job evaluation if she persists. The coop student has exhausted the other options: gathering information, proposing value integrating, compromising, and trade off decisions. The only choice now is to support the immoral dictates of the group leader or oppose his decision. Opposition opens a new series of action alternatives. The coop student could talk with the other group members to see if she could get them to appeal collectively this decision. Or she could go outside the group to a high supervisory level and inform this person of the problem. She could consult a corporate ombudsperson, the human resources department, or the personnel department, whichever is appropriate for this organization. In short, she could pursue and—time permitting—exhaust internal channels of ethical dissent. Should these fail, external opposition would be the last resort; she could go outside the organization to a governmental agency like OSHA and blow the whistle on her coworkers. Each of these forms of opposition has its strengths and weaknesses. Each brings with it certain risks.

Putting all of this together we can generate a set of rules or guidelines for integrating ethical considerations into practical decisions. When confronted with a problem that has an ethical dimension, say, that places an important ethical value in jeopardy, we can use the following guidelines to design a solution:

1. Try to dissolve the problem by gathering more information
2. Aim first for designing a value integrative solution where the conflicting values are fully integrated and realized.
3. Should a value integrating solution not be available, design a value compromise where each of the conflicting values is partially realized.
4. Should a value compromise not be attainable, prioritize the conflicting values and choose to realize the most important value from an ethical standpoint setting the other value off to the side.
5. Should none of these solutions be available, and the decision is immoral, pursue available forms of opposition starting with internal opposition and turning to external opposition only as a last resort.
6. Finally, should even opposition prove unacceptable, give careful consideration to exiting from the situation to avoid being implicated in an immoral project.

In general, our approach to this stage is different from the traditional approach to problem solving in two important ways. In the traditional approach, we evaluate existing alternatives in terms of ethical considerations to find out which is best. In this approach we begin with the realization that solutions are not found but must be designed. Most important, when we design solutions to problems, we must make special effort to integrate the conflicting considerations. Ethical considerations thus play a more active role in problem solving; instead of evaluating ready-made solutions, they help guide the process of creatively designing new solutions.

Second, we assume that value integration represents the optimal problem solving strategy. We always aim first to design a solution that integrates conflicting values, fully integrating each. Notice we are not claiming that all ethical problems can be solved by integrating values. Rather, we are saying that value integrative solutions are the best ones and that we have to make a serious effort to construct them first before resigning ourselves to value compromises, value tradeoffs, or opposition strategies. This is a consequence of following a virtue strategy that asks, "What is the best I could do in this situation?"

Solution Testing
Problem specification provides one way of testing the solutions generated in the previous stage, that is, we can evaluate solutions in terms of how effectively they solve the problem. If the solution integrates the conflicting values identified at the problem specification stage, then that is a telling factor in its favor.

To provide further testing, we propose four ethics tests: reversibility, harm/beneficence, pubic identification, and a code of ethics test. They provide a comprehensive standpoint from which to examine a proposed solution. The reversibility test examines the solution in terms of its formal characteristics. Harm/beneficence examines the solution in terms of its results or consequences, sorting these out into benefits and harms and weighing the one against the other. Public identification turns from the action itself to examine the agent. The action reveals the character traits of the agent, his or her virtues or vices.

Putting them together, we accomplish a comprehensive review of the solutions we have designed, a review which helps us to improve our solutions, evaluate them, rank them, and choose from them the best available course of action. The tests also provide us with a means of explaining and justifying our action to others.

Reversibility
This relationship-oriented test is rooted in the philosophical perspective that asks about duties we owe to each other. It takes advantage of the "universilizability" aspect of Immanuel Kant's formal approach to ask how the action treats other rational creatures. The primary question this test asks is, "Would I think this a good design or research project were I among those affected by it?" To set this test up one must first identify the agent, i.e., the person performing the action and then describe the action or solution under consideration. Then one identifies the action's stakeholders, i.e., those who stand to be affected by it in an important or essential way.

Then comes the piece for which all the set up was done: Reverse with each stakeholder by putting yourself in his or her place and asking if you would still think the action acceptable were you on its receiving end. Doing this reversal requires moral imagination. After evaluating the action in terms of all the stakeholders, think through conflicts that arise between stakeholder standpoints by devising strategies for treating people with respect.

Harm/Beneficence
This outcome-oriented test is grounded in the philosophical approach called utilitarianism, in which the primary question is "Do the benefits outweigh the harms associated with this course of action?" Again, one first identifies the agent, and then describes the action including the motive, goal, the expected results, and their magnitude. One then sorts these expected consequences into harms and benefits weighing the one against the other. Then one compares the action with other actions and looks for the solution that maximizes benefits and minimizes harms. Finally, one checks for inequalities in the distribution of harms and benefits (e.g. are some people only harmed?).

Public Identification
This social-reputation based test is grounded in the virtue ethics approach that is central to this text. It asks the question "What does this act tell us about the character of the agent?" Again, one identifies the agent and describes the action. Then, associate the action with the agent using it to come up with a character sketch of the agent. Finally, use the list of virtues we provide to help fill in this character sketch.

Code of ethics test
This professionalism-based test is grounded in the work that many computer professionals have done on codes of ethics that can be used to guide the evaluation of professional activities. Again, identify the agent and describe the action. Then evaluate the action in terms of the following important values in codes of ethics:

- Does your action hold paramount the health, safety, and wellbeing of the public, i.e., those affected by computing projects outside of the professional client relationship?
- Does your action maintain faithful agency with the client by avoiding conflicts of interest and maintaining confidences?
- Does your action promote the autonomy and reputation of the profession?
- Does your action maintain collegial relations with computing, professional peers?

Together, these tests provide a comprehensive view of the actions or solutions you might consider. Most of the time, they are consistent with one another. Often they converge on one or two solutions; this convergence should be taken as an additional sign of a solution's strength, a sort of meta-test, if you will. When the tests disagree, this should not be taken as a sign that one should be retained while the other tests are set aside. Rather, conflicting information from the tests clues us into the weakness of a given solution. If possible, we should alter the solutions to remove these conflicts or combine elements of different solutions (each strong in relation to a given test) into a new synthetic decision. In short, the tests are not algorithms that allow us to crank out perfect solutions. Rather they give us insight into the strengths and weaknesses of our solutions and thus allow us to design better ones or to anticipate problems that come when we turn to implementing our solutions in the real world.

Solution Implementation
Carrying out this stage requires that we anticipate obstacles and develop plans for overcoming them. To do this we apply a fifth test to the problem solving process, a non-ethical, feasibility test. In short, we ask in a careful and thorough manner if and in what way our solution can be realized in the real world.

This can be put in a slightly different way. In the first stage, problem specification, we identify and organize those requirements to which a good solution should respond. We identify the values embedded in socio-technical systems, how these are supported by the elements of the STS such as hardware, software, physical surroundings, people/groups/roles, procedures, laws, and data plus data structures. These constituents of the STS frequently embody values that conflict with one another. Problem solving requires that we design solutions that integrate these conflicting values or find other ways of making them compatible. We also gain insight into solution specifications by testing prototypical solutions with the ethics and code tests. But design solutions do not only embody and integrate these specifications; they also must do so within certain background constraints. The purpose of the solution implementation stage and the feasibility test that accomplishes it is to identify and respond to these background constraints.

We can identify the following seven constraints that help us to identify obstacles to integrating our solutions in the real world:

1. *Time*. Is there a deadline within which the solution has to be enacted? Once we identify the time constraints, we have to examine whether these are fixed or

negotiable. If we need more time to implement our ethical solution, then perhaps we can negotiate an extension of the time limits.

2. *Financial*. Implementing solutions cost money. Are there cost constraints on implementing the ethical solution? Can these cost constraints be extended by raising more funds? Can they be extended by cutting existing costs? Can we negotiate more money for implementation?
3. *Technical.* Technical limits constrain our ability to implement our solutions. What, then, are the technical limitations to realizing and implementing the solution? Could these technical constraints be moved back by modifying the solution or by adopting new technologies?
4. *Manufacturability*. Are there manufacturing constraints on the solution at hand? Given time, cost, and technical feasibility, what are the manufacturing limits to implementing the solution? Once again, are these limits fixed or flexible, rigid or negotiable?
5. *Legal.* How does the proposed solution stand with respect to existing laws, legal structures, and regulations? Does it create disposal problems addressed in existing regulations? Does it respond to and minimize the possibility of adverse legal action? Are there legal constraints that go against the ethical values embodied in the solution? Again, are these legal constraints fixed or negotiable?
6. *Personnel.* The STS within which the solution is to be implemented contains people, the roles they play, the groups they form, and the procedures that coordinate their activity. Are there elements of these features that could oppose the implementation of the solution? For example, could a stubborn, recalcitrant supervisor oppose the solution out of sheer spite? Does this solution or its implementation require a change in organization climate or standard operating procedures? Once again, are these opposing features in the STS fixed or flexible?
7. *Social, Cultural, or Political*. The STS within which the solution is to be implemented contains certain social structures, cultural traditions, and political ideologies. How do these stand with respect to the solution? For example, does a climate of suspicion of high technology threaten to create political opposition to the solution? What kind of social, cultural, or political problems could arise? Are these fixed or can they be altered through negotiation, education, or persuasion?

This feasibility tests asks us to consider constraints. How could these hinder the implementation of the solution? Should the solution be modified to ease implementation? Can the constraints be removed or remodeled by negotiation, compromise, or education? Or can implementation be brought about by modifying both the solution and changing the constraints? The feasibility test provides a good way of anticipating problems and helps channel our thought in responding to these problems.

You may have noticed that some iteration in this model has already been implied. Solution generation may send one back to problem specification to investigate a newly important part of the socio-technical system. Solution testing and implementation often involve modification of solutions, thus sending you back to the solution generation stage. And like circular software development life cycle models, once you implement the solution, you need to start over again as a part of maintenance of the software.

Integrating the model in software development.
One recent discussion of software development process (Sawyer ACM ref) categorizes the processes into three different types: sequence, group, and network processes. The ethical problem solving model will look a little different in each of these cases.

Sequential processes
The sequential model is the traditional, linear, structured approach to software development that is most often used in industry. There are many implementations of it, including the software development life cycle (ref) and the Software Engineering Institute's Capability Maturity Model (ref). Common to them all is an initial stage of *feasibility testing* or *business modeling* to determine if the project is correctly conceived, compatible with the goals of the organization, and financially feasible. It is at this point that initial work on specifying the socio-technical system can help identify value conflicts that make the project a poor match for the organization. A common second step is *requirements modeling*, in which the functional requirements of the system and the constraints on the system are identified in detail. Much of the work one does in requirements analysis is related to work done to specify the socio-technical system, and so these steps can go hand in hand here. *Architectural design* is a common third step, in which the overall design of the system is specified. Decisions made at this stage can affect a wide range of stakeholders, so stakeholder analysis and ethical evaluation are crucial during this stage. *Implementation* involves pulling together all the pieces of the system and testing them to make sure they perform as required. At this stage, ethical analysis may shift to decisions made about the process itself (e.g. how much testing is good enough?). *Deployment* is the actual rollout of the software to the customer. At this stage numerous ethical issues arise since one can more clearly envision the effects of the system in the organization. In addition, the process of deployment itself raises ethical questions about its effects on the organization. Most sequential models also have a *maintenance* phase in which corrective, adaptive, perfective, and preventive maintenance are done. In this stage one is really involved in a series of smaller projects, and so ethical problem solving across the entire scope of the model may come into play.

The sequential model usually has a management team of software engineers and others who guide the effort. It is these who should be responsible for seeing that ethical problem solving is integrated into the process. In the same way that they may hire specialists to deal with particular aspects of the project (e.g. networking, interface design) they may want to hire specialists to help them think through the social and ethical implications. Many of those involved in this approach are confined to very specific roles and never interact with the larger socio-technical system for which the system is intended. This puts a special burden of responsibility on those who are in charge to see that an eye is kept on the larger context.

Group processes
Group processes of software development focus more on interactions among the members of the development team and interactions of the team with customers or users. Software is often rapidly prototyped and goes through numerous iterations to improve it

until acceptable standards are reached. This usually means that the boundaries between roles in the team and between the team and customers/users are quite permeable. Key stakeholders are often directly involved in development.

Since there is much role switching in this model, all the members of the development team need to be aware of the ethical and social issues associated with the project and need to keep an eye on how their decisions in the rapidly changing design affect the values of the larger socio-technical system. Still, in this approach, there are almost always identifiable early, middle and late periods where different kinds of ethical issues will arise, just as in the sequential model.

Network processes
The most common example of network processes of software development is open source software development. This is a product-oriented approach, and often there is no clearly identifiable "team" other than the central group of chief programmers or decision makers. Additions and modifications of the system are contributed by a loosely organized group of programmers and the central team makes decisions about whether to incorporate the suggested revisions or not. In this process, both the central team and the individual programmers will have ethical roles to play. Individual programmers may themselves notice a value mismatch and attempt to devise a solution for it, submitting it along with an argument about what problem it solves and why it is effective. The central team will also need to keep an eye on the bigger socio-technical system and the effects of the software on it. Again, in this approach there are almost always identifiable early, middle and late periods where different kinds of ethical issues will arise, just as in the sequential model.