(human) execution of design and production is faulty. E.g. coding slips, a program that does not meet its requirements. Possibly, but questionably, this could be modelled statistically, depending on how good a statistical model of the generation of human errors in intellectual tasks we can build.
3) Uncertainties and inadequacies in the design method itself. In particular, the absence of any method to guarantee that all relevant requirements are identified and written down.

How to test for the unforeseen, for missing requirements? The best approach may be to see it working. Hence in practice, as opposed to theory, the first time test may have a special importance, which would explain why engineers including programmers always want to test their creations out, usually informally "just to see if it works".

The first time you see a version of your program working is a heartening sight. And more and more you see justifications of development methods that allow this early on e.g. by writing stubs for all parts. There is a sort of reason for this in that as soon as it runs, however stubby, you can use the program output / behaviour as an additional source of information. But behind that is a reason something like this: If the program runs at all, then the logical AND of a very large number of required properties must be (or is very likely to be) true, so a huge leap in certainty is made in one test. (In contrast, later more systematic tests are mainly addressed at seeing that the scope of the assertions is true: that it goes on working as various parameters are pushed to the limits of their ranges.) But a further reason lurks there: that not only the AND of the explicit requirements must be true, but so must the AND of all the infinite implicit requirements. In other words, this first success is also a test of the most uncertain aspect of the whole design process.

Thus the most informative test is probably the first informal one — the one done intuitively by all programmers, but seldom mentioned in "methods".

The problem with the Hubble space telescope would of course have been detected if only they had done this test: any test of the whole assembled telescope. It was not a random problem, one of uncertainties in materials. It was a problem (of type 2 in the list above) within a module (shaping the main mirror) that affected implementation and testing equally. It would have been detected by a whole-system test, because that would have shown what that module looked like to other modules.

I have constructed an argument that the first informal test of an artifact, for instance a program, has a special importance. Is this true? This suggests a research project, which I haven't done: to keep a record of when a given programmer discovers each bug/problem, and so discover the proportion that a) were discovered during informal rather than formal tests; b) to discover the proportion that could not have been discovered in formal tests because they were bugs in the requirements or specs, not in the implementation.

References

Berry,D.C. & Broadbent,D.E. (1989) "Problem solving and the search for crucial evidence" Psychological research vol.50 pp.229-236

Berry,D.C. & Broadbent,D.E. (1990) "The role of instruction and verbalization in improving performance on complex search tasks" Behaviour and information technology vol.9 pp.175-190

Gilmore, D.J. "Visibility: a dimensional analysis" in HCI'91 People and Computers VI: Usability Now! (eds.) D.Diaper & N.Hammond pp.317-329 (Cambridge University Press: Cambridge).

Petroski, H. (1982) To engineer is human: the role of failure in successful design (Macmillan: London).

3. Unplanned testing and bug detection

A job requirement for many programmers is to design and perform systematic testing on software e.g. acceptance tests. There is a body of knowledge purporting to be about how this should be done (e.g. the concepts of black box and white box testing). This report argues that in practice some of the most important discoveries from testing are not covered by these ideas, and that therefore different skills and different approaches are required, and presumably actually practiced by successful programmers, than those described in the literature.

In brief: the most important discoveries from testing are the unexpected issues that are obvious to a human observing, and that correspond to missing requirements. However not only are these not deliberately sought, but they threaten software engineering as a rational enterprise: how could anybody plan rationally to discover the unexpected?

In fact this is nothing special about software engineering. Petroski's books argue that civil engineering, for instance, progresses in part by learning from disasters which mainly reflect, not negligence, but learning the hard way about new requirements that were always implicit and automatically satisfied until old parameter ranges were exceeded and they emerged into significance. A simple example would be, that if you build bridges out of stone then you need never worry about side winds as by the time you have satisfied the requirement to carry the vertical load the structure will be too massive to be affected by wind. With modern steel bridges this is not the case, as was discovered when the Tacoma Narrows bridge disintegrated. Since all designs depend on an infinite number of requirements, these can never be written down and checked (the bridge, or the software, must work at all phases of the moon, when the operator drinks tea, if someone speaks Chinese nearby, if the wind blows at 47 m.p.h., if the wind blows at 41 m.p.h which just happens to correspond to the resonant frequency of the artifact, ...). All rational design can do is to write down the requirements that previous experience suggests may not be automatically satisfied; but how do you guard against the unexpected, against a new requirement becoming important for the first time in this case? You build the artifact, and you try it out i.e. test it. If nothing undesirable happens, then it is probably OK. But you cannot be sure (perhaps the side wind didn't blow on the day of the test), and you cannot design these tests by considering the explicit requirements and specs, because what you need to detect is the issues missing from those lists.

Examples in programming might be noticing that the software runs too slowly, that when an error message appears it obscures the display it refers to, that the most common user error in selecting an input file is that the same file is still open in another editor and special support for this should be provided. Any problem, once identified, can become part of a standard set of requirements to be applied in future to most or all projects; and in principle this should happen. However firstly, there must be a first time the issue occurs; and secondly, in practice such requirements often are not written out, but rediscovered by programmers during testing. This rediscovery is NOT because programmers explicitly foresee this possible error and then test for it. Rather, they "just notice" what the problem is when they run the program.

Can we think about testing rationally? Programmers are usually taught about black box and white box testing. Actually these concepts are undermined by similar issues. Black box and white box testing are really the same in that they both depend on (possibly unjustified) assumptions about the device in order to get a few tests to stand in for the huge number really needed to be exhaustive. Black box tests typically assume that if inputs and outputs are, in mathematics, continuous ranges of values then the implementation will be smooth in some way (so only a few values need be tested). This of course may be wrong e.g. if lookup tables are used.

Basically most testing is for foreseen errors. This requires two theories: one of how the device should work, the other of how errors are generated. Together they predict how to detect errors. White box theories explicitly use a theory of how a device works, and black box theories implicitly use such a theory in assumptions about the nature of the input and output functions. What kinds of theory do we have about how errors / bugs are generated? Testing checks on uncertainty in a product's properties. There are 3 broad sources of uncertainty:

1) Unreliable elements in the products e.g. metal fatigue, operator error in human-machine systems. In principle this can be treated statistically.
2) Errors in executing the design and engineering process: where knowledge is adequate, but

6. Diagnostic testing: the skill of designing input data to explore and diagnose a bug given a preliminary symptom. Our students are now required to perform tests and submit documentation of this, but it is clear that they do not expect the testing to discover any bugs. Consequently their tests look plausible but lack real diagnostic power. The exercise gives them executable programs (no source: true black box), a brief statement of the program's function, a vague description of a problem and the original test data giving rise to suspicion, and the task of designing test data to refute or sharpen up the suspicion.

7. Black box tests. Given an executable program file, and a brief description of its function (built into the program and displayed on each run), the student must design a set of tests to discover what problems if any it has.

2. Visual layout of tables to support problem solving

Berry & Broadbent (1989, 1990) studied a problem solving task based on using printed tables of data. The task is to play the role of a river inspector who has to decide which company is responsible for a pollution incident. A table lists the unique combination of chemical pollutants each company uses, and the task is to request in sequence a series of tests until the company responsible can be determined. As tests cost money, the best solution will minimise the number of tests needed.

Berry & Broadbent were mainly concerned with what strategies people used, and how they could be trained in the optimum (binary split) strategy. In fact people are in many cases very resistant to using the optimum strategy, even when given direct training. This inability to use the best strategy seems to be due to the layout of the table given to subjects, which in their experiments consisted of a list of factories, and against each factory name, a list of pollutants.

Gilmore (1991) ran variations on these experiments. His purpose was to analyse an apparent cognitive dimension of "visibility" into three dimensions, which he named accessibility, salience, and congruence. He compared four table layouts by varying a) whether the tables gave factories first then pollutants against factories, or vice versa; b) whether the secondary properties (e.g. pollutants) were given as a list or in a grid so that a reader could easily scan for all the primary instances (e.g. factories) that shared a given property (e.g. pollutant). Gilmore showed that:
a) Different table formats vary the difficulty of carrying out any given method; and conversely the usefulness of a format depends on the method used.
b) Different table formats vary the difficulty of the task (i.e. of the best method for the task, given the format).
c) The method chosen by the user depends on the task but also on the user.
d) The method chosen by the user, and whether they choose the optimum procedure, is affected by another property of the format ("salience"), largely independent of features determining difficulty. I.e. what procedure seems obvious to users is also, but independently, influenced by table format, and this is often independent of any explicit training given to subjects.

These tables are in effect a visual notation for supporting a task. The format of these tables, then, can be varied in a number of ways including: by which of the two entities (factories or pollutants) is primary, by whether lists or a 2D grid layout is used (i.e. whether columns are meaningful), by whether each of the dimensions has random order, alphabetic ordering, or some other ordering. Berry & Broadbent fixed on one format and studied how users could choose a method for the task given the format. Gilmore compared formats, showing effects on choice of method and on the effectiveness of a chosen method, and hence on task performance. However it is interesting to consider an alternative task: not how to choose each test for pollutants in turn, nor how to choose a method for that task, but how to make reformatting choices for the table in order to make the task easier: the corresponding visual notation selection task.

In the talk, I will illustrate some of these alternative formats, and also (by asking the audience to suggest modifications to the current format) that we are actually poor at choosing a better or optimum format for the task.

A set of additional component programming skills
My provisional set is as follows. This is in addition to the identified skills of:
A. Implementation
B. Design and structuring of a program
C. Documentation
D. Acceptance testing i.e. demonstrating it works.

1. Lexical issues. These students have never seen an un-indented program (the environment does this), or one with completely uninformative identifiers (they are not forced to work with other people's code, let alone with perverse naming systems). The only exercise I have provided is a printout of a working program with formatting removed and identifiers randomly renamed. This is probably not effective, as it is not an activity (beyond having the students feed it to the THINK system and see the indenting recreated).

2. Learning what problems typically lie behind error messages. A major part of the expertise a tutor can bring to a student with a problem is of the kinds of bug associated with each error message. THINK's error messages are not particularly good from a programmer's viewpoint. In most cases, the content can be safely ignored and the programmer just looks at the line indicated to try to spot any obvious syntax violation. The second class of messages do describe the problem usefully e.g. "x is not declared", and require no training. However there is a third class where the message actually is quite diagnostic to an expert, but not to a novice, and learning is required. For instance "Insufficient stack space to invoke procedure or function" is caused by infinite recursion, "Your application zone is damaged" is caused by having procedures return huge values (e.g. large arrays: using a var parameter would avoid this), "Bus error" usually means an error with using pointers, and having the whole THINK application crash usually means that there was an array subscript error without bounds checking being turned on.

The exercise tells students that they should, as part of learning each new Pascal topic, deliberately provoke errors in order to learn the error messages, and supports this with some example error-provoking code.

3. Finding bugs by reading. Although these students are sometimes required to read code, they do not seem to take this seriously as a debugging method. Whereas in the days of slow batch compiling, reading printouts of code was an important activity, we probably have to teach it nowadays. The exercise requires them to find some bugs by reading only. What we have to realise, and then to teach the students, is that some bugs are easy to find by reading and it is inefficient to look for them in other ways; but other bugs are hard to find this way and other techniques are better e.g. testing and using debugging tools. There are in fact 2 skills here: a) spotting errors that do not require understanding of the program e.g. missing declarations, redeclarations of the same identifier, code that can never be executed. b) Predicting what the code must do, and realising that this is not (cannot be) what is wanted. E.g. looking at a set of conditions (e.g. in a multi-branch if statement) and deciding if cases seem to overlap or be missed.

4. Debugging tools. We seem to need to train students in using the debugger. They do not seem to teach themselves, even though they need it at times, and even though I didn't have trouble teaching myself how to use it without documentation. Probably what they lack are the ideas of what would be useful if it were provided, and what is likely to be provided in a modern debugger. E.g. being able to inspect the value of any data location at the point the program stopped, the calling sequence, being able to tell the program to halt at some specific point in the code. The exercise gives them a basic tiny manual for operating the debugger, and some buggy programs whose bugs they must find.

5. Spotting symtoms in Input/Output samples. As a prerequisite for doing testing, students have to be able to notice that a program's output is not in fact correct. Many students do not notice this: as if anything that did not provoke an error message must be running correctly. The exercise is reading again: reading not code but printouts of input/output samples. For instance the first exercise says "The program reads in a list of names sorts them; and prints them out with some processing" and gives a sample and the students have to spot that one of the input data lines has been omitted from the output (a typical bug in a sorting routine).

1. Teaching programming's component skills

For several years I have assisted as a tutor on a first course at MSc level on Pascal: for most students it is their first programming. It uses THINK Pascal on Macintoshes. This environment does automatic indentation and layout of programs (e.g. putting syntax words like "while" into bold), it has a single command that checks, compiles, links, and runs a program if possible, halting with error messages as necessary. It also has a rather good debugger with multiple subwindows, inspection of all variables including any component of complex data structures, insertion of halt points, etc. Students' experience of programming, then, is to rough out a design on paper (or take a model program from tutorial handouts), type it in to a visual editor, try and run it reacting to error messages from the compiler, and then when the program runs they can type input data to their program online, or prepare a data file in an editor.

The course has been iteratively refined over many years. In prehistoric times, courses may have revolved from the students' viewpoint around getting programs to run, with a consequent prevalence of horrible code and ad hoc changes. However, at least on this course, this has long been submerged by the aim of teaching structured program design. Problem solving gets as much time in lectures as programming language features; marks from very early in the course onwards are given as much for good modular design and documentation of high level design (so called "level 1 plans") as for getting the program to run. This is now so successful that it is several years since I have been asked to look at a program that is hard to understand because it is horribly structured or laid out. Current improvements in teaching are aimed at making testing and documentation as important as design and implementation. (These improvements are still being worked on: they are reflected in the marking scheme, but only partially in student practice as they do not yet have their full share of lecture time and paper materials to support the students by precept and example.)

However my experience suggests that these impressive teaching achievements have a down side: that students currently show surprising deficiencies in the area of debugging. In earlier times, students acquired considerable skill at debugging because success largely depended on reacting to problems by modifying the program until it worked i.e. on debugging. Although it wasn't taught directly, student success depended on this skill and they acquired it through practice. Now it seems they can be quite deficient. They are successfully taught top-down structured design, with the result (just as in the claims for this method) that many problems do not arise in the first place. Furthermore, because Pascal is a strongly typed language, most low level errors are caught by the environment and the error messages point to them fairly clearly. When the occasional bug occurs that does not fall into this category, the students often seem helpless, and unaware of even basic debugging ideas such as inserting tracing code to show how far the program gets before failing or what the values of variables are during a run, or how to go about discovering what an obscure error message really means.

My conclusions

Since these skills are far more transferable (to other languages and environments) than Pascal syntax, and even than structured design (since in many jobs you have to troubleshoot other people's code not design your own), this may be serious from an educational viewpoint. This experience draws attention to the fact that "programming" is in fact a multi-part skill, and that most parts of it are not usually taught at all. Hitherto students were forced to learn them through practice, driven by the need to get programs to work. As both teaching and programming environments have improved, we have reached the stage where we need to identify these component skills explicitly and to consider how to teach them explicitly. I have identified such a list provisionally, created some tutorial exercises for each one, and trialled these informally on students. However they are not yet an integral part of the course and so have not been properly educationally tested.

In other words, "programming" in fact requires a diverse set of skills, reflecting the many component tasks of the overall activity of "doing programming". The environment modifies what the tasks and skills are, for instance by relieving the programmer of some jobs. In a teaching context, this can mean they fail to learn some skills (possibly the most transferable and important skills). Well designed teaching, too, can have a considerable impact on both the skills learned, but also on skills not required as the result of good practice removing much of the need for them. I suggest that we need to identify explicitly all the component skills a programmer should have, and to design teaching for each of these skills instead of merely expecting them to be acquired implicitly through practice of the overall activity of programming.

Programming skills, visual layout design, and unjustifiably useful testing:
Three reports in the psychology of programming

by
Stephen W. Draper
GIST (Glasgow Interactive Systems cenTre)
Department of Psychology
University of Glasgow
Glasgow G12 8QQ U.K.
email: steve@psy.gla.ac.uk
WWW URL: http://www.psy.gla.ac.uk/~steve

Preface
This is a draft of a talk for PPIG96 10-12 April 1996, Ghent, Belgium

Abstract
1. Teaching programming's component skills
"Programming" in fact requires a diverse set of skills, reflecting the many component tasks of the overall activity of "doing programming". The environment modifies what the tasks and skills are, for instance by relieving the programmer of some jobs. In a teaching context, this can mean they fail to learn some skills (possibly the most transferable and important skills). Well designed teaching, too, can have a considerable impact on the skills learned, but also on skills not required as the result of good practice removing much of the need for them. I suggest that we need to identify explicitly all the component skills a programmer should have, and to design teaching for each of these skills instead of merely expecting them to be acquired implicitly through practice of the overall activity of programming.

2. Visual layout of tables to support problem solving
Further study of work by Berry & Broadbent and by Gilmore on a problem solving task based on data presented in tables uncovers a new kind of task: one of how to select a visual notation (i.e. one of the possible table formats) to make a task easier. The earlier work showed strong effects on how easy the problem task is depending both on the table format and on the method a user selected to solve the problem. The talk will attempt to demonstrate that we are not good at redesigning the table format to make the task easy.

3. Unplanned testing and bug detection
An argument is constructed that informal first time testing of a progam, just to see if it runs, has an importance out of all proportion to its lack of theoretical justification or place in official methods. It is an open empirical question, requiring research still undone, to establish how many bugs are detected at this stage, and whether they have a special nature different from those which black box and white box tests are designed to detect. My argument is that it is missing or implicit requirements that are of this kind, and that their detection requires open ended observation, not controlled testing. This is analogous to the contrast between hypothesis testing by controlled experiments versus field studies that are able (but not guaranteed) to go beyond the preconceptions of the investigator and detect the unexpected.

Introduction
This presentation will consist of three short reports on quite different topics in the psychology of programming.