Empiricism in Software Engineering: A Lost Cause?

Essay for MNVIT401 2002, by Marek Vokac

Final version

Present-day natural sciences are solidly founded on principles relating knowledge to observable quantities. As I shall review, the relation has been present to some extent from Antiquity via the Renaissance, Enlightenment, Logical Positivists and Popper to the present day. The *presence* of this relation has not been a matter of serious dispute in the natural sciences for a long time.

Much of the subject "Software Engineering", in the Computer Science discipline, seems to be an exception to this rule. There are methods, processes, organisational theories, architectural patterns and programming styles and guidelines galore with little, if any, reference or justification in empirical observations or experiments – even in the most prestigious, academic journals.

What are the reasons for this state, why is it important, and why is it so very hard to do anything about it?

The historical foundations of modern natural science

Modern natural science has deep roots in Antiquity. In this essay I will only touch those points along the evolution that I deem important to the argument.

The Greek period

Conventionally, we trace modern natural science back to the Greek era. The Greek philosophers represented a break with religious tradition in that they sought the answers to fundamental questions in nature itself, without invoking an external spirit as creator or prime mover[1].

The concepts of truth and reality as something to be discovered, described and understood by Man was also a central feature. Though thinkers such as Aristotle, Plato and Democritus disagreed on many fronts, the principle that one must observe in order to learn was never in doubt. To what extent our senses are to be trusted, and/or extended by devices was another matter.

Another notable change was taking practical geometry, and restating it as a formal, theoretical system of axioms and proven theorems. The fact that one can make a right-angled triangle using a rope with 12 evenly-spaced knots was known to the Egyptians and Babylonians, but it took Pythagoras to state the fundamental rule in mathematical terms[2]. A crucial relation between abstract, formal methods and the observable world was thus established as possible and even desirable.

It is notable that those parts of Greek science that survived the test of time were those that were based in some part on observation, or were purely abstract. The geometrical theorems were empirical in their basis, while discussions of what might be the elementary substance were less so. The more mystical aspects of the Pythagoreans only survive as curiosities, not as science.

The Middle ages

In the Middle ages mathematics prospered under Muslim patronage, with important contributions such as algebra. In Europe, most of science degenerated into discussions based solely on ancient texts and religious viewpoints. While this certainly makes interesting reading, the impact on enduring natural science was minimal.

From a modern viewpoint it might seem obvious that relying on an ancient text while studiously avoiding reality hampers progress. People were also hampered by the fact that many of the ancient texts were lost. The Library of Alexandria was burned by the Christians in AD 390 and the Muslims in AD 641[3].

From around AD 500 to 1200 the Encyclopaedias came into being. By saving a recopying old and often fragmented knowledge, they at least halted the loss and laid the foundations for future progress.

Around the end of the first millennium AD, monasteries had become repositories of knowledge and intellectual activity. The feudal system, like the slave-based economies of the Greek and Roman eras, ensured that there were at least some people who did not have to spend all their time maintaining their existence. From 1200 we can see the first universities come into being.

However, much if not most intellectual activity in Europe was centred around the Church and its needs. Augustine maintained that science and faith can coexist, but this was never an easy coexistence. Thomas Aquinas managed to combine faith and Aristotelian science, but proving the existence of a God was beyond him.

Renaissance

The case of Giordano Bruno illustrates the problems of attempting to combine science, based on observations and free intellectual endeavour, with the Church and its faith. Bruno was far from being an atheist, but his worldview broke with the heliocentric, Aristotelian orthodoxy. Add to that his criticism of the Church itself, and the result was probably unavoidable. Bruno spent 8 years being tortured in prison, and ended his life at the stake[4].

Copernicus, Kepler and Galileo embodied the principle of basing theories and descriptions on observation, *even if* the explanations thus obtained did not agree with orthodoxy. While they were not always correct, it is reasonable to say that their methodological stance was quite modern and certainly empirical in its foundations.

Enlightenment

From the early 1600's natural science made great leaps, aided by people such as Euler (math), Boyle and Huygens (physics), Dalton and Lavoisier (chemistry), Harvey (biology) and Linné (botany). Observation, theory and experiment were the rule rather than the exception, and much of the knowledge gained still survives in refined form.

An excellent example is Harvey's discovery of the blood circulation. He combined earlier knowledge about the one-way valves in the blood vessels, with his own observations on 40 species. In cold-blooded animals the heart contractions continued for some time even after the heart was removed, and this enabled him to recognise the heart as a pump for blood.

Lavoisier published a catalogue of 24 atomic elements in 1789, and by experiment proved that water is not an elemental substance. Observations on the other Greek "elements" (fire, earth, air) showed that they were also not elemental, but combinations of other substances.

The positivists

The Logical Positivist movement was centred round Vienna in the years leading up to World War II. They sought to relate all natural science to observable facts, and thus to *prove correctness*, at least in an ideal sense. The movement was also motivated by politics and the need to distinguish "real" science from various pseudosciences, such as the racial ideologies espoused by the Nazis.

This was a serious attempt to define *how* natural science should be done, and to justify that scientific knowledge – knowledge that we arrive at using a *scientific method* – is somehow objective, valuable, and can be separated from other knowledge in a relatively objective manner.

Popper

Positivism runs headlong into the problem of *induction*, the jump from "every time we looked using such equipment as we had at that time" to "always, everywhere and for all of eternity". Strictly speaking, such a jump is justifiable only within the confines of a formal system such as mathematics (and even there, Gödel's Theorem tells us that within any formal system complex enough to be interesting, there will be improvable truths[5]. One can always define a wider system, but the process is not bounded…).

Karl Popper turned to *falsifiability* of a theory as the property to be used in the scientific model of seeking the truth:

"Although we have no criterion of truth, and no means of being even quite sure of the falsity of a theory, it is easier to find out that a theory is false than to find out that it is true (as I have explained in detail elsewhere). We have even good reasons to think that most of our theories-even our best theories are, strictly speaking, false; for they oversimplify or idealise the facts. Yet a false conjecture may be nearer or less near to the truth.

...

By incorporating into logic the idea of verisimilitude or approximation to truth, we make logic even more 'realistic'. For it can now be used to speak about the way in which one theory corresponds better than another to the facts-the facts of the real world."[6]

Of course, this is based on the premise that it is actually meaningful to talk of some kind of objective reality. However, one should not make this assumption stronger than actually required. All we need is that, under the same circumstances, measuring the same phenomenon will give the same result for people in different places and times. On this premise we base modern natural science, extending its reach out into space (through telescopic observations) and back in time.

The case for empiricism

The history of science contains overwhelming evidence for the view that scientific knowledge must be based on some kind of observation or experiment, performed in a

systematic manner. And observations themselves are not enough; we can only talk of understanding or explanation if we can reduce multiple observations into a coherent model. Further, we require that his model give rise observable consequences, which we can then use to refine it. It's a never-ending process.

When we look back on history, we the accumulated collective knowledge of humanity consists of those facts and models that have some root in observation and empiricism. The rest is a matter of *belief* and properly the domain of religion, or curiosities we remember because they make good stories. If we restrict ourselves to look at technology and its basis, belief will not take us very far; arguing that something "should" work has never made much impression on nature. It seems reasonable to conclude that solid, useful knowledge is inextricably linked to an empirical basis, and any Science worthy of the name has to take this into account.

Aside: Beautiful experiments

In May 2002, Robert P. Crease invited nominations for the most beautiful experiments in Physics, presenting the results in September in PhysicsWorld[7]. All of these experiments demonstrate how relatively simple observations can unlock extraordinary insights into Nature. All of them illustrate the power of the **experiment** as a tool of science.

Eratosthenes and the diameter of the Earth

When Eratosthenes heard of a place where things have no shadow on midday of summer solstice (Aswan), he realised that the Earth's size could be measured. On the same day and time he measured the amount of shadow cast in another town directly north of the first (Alexandria), and found that there was about 7 degrees of slant from the vertical.

The rest was geometry – 7 degrees is about $1/50^{th}$ of a circle, so the size of the Earth should be 50 times the distance between Alexandria and Aswan. We do not know the absolute length of the units he used, but the idea is sound and he should have had the correct order of magnitude.

Galileo and falling objects

During Galileo's times in the late 1500's, it was obvious that heavy things fall faster than light ones – and Aristotle had said so. Using lead bullets dropped from the leaning tower of Pisa, Galileo conclusively proved otherwise. Apollo 12 astronaut [ref] did so too, dropping a hammer and a feather on the moon.

Newton's decomposition of sunlight with a prism

Around 1680, white light was held to be purer and cleaner than any other. Isaac Newton's simple act of placing a glass prism in the way of a beam of sunlight showed how it is instead composed of a mix of all "colours", and that whiteness is simply something we assign to that mix. Later research has expanded the electromagnetic spectrum many octaves in each direction, and shown how the eye relies on a limited set of bands to produce our perception of colour.

Cavendish's torsion-bar experiment

How strong is gravity itself? Henry Cavendish understood how it could be done. He placed small metal balls at each end of a six-foot wooden rod, suspended in the middle so it could rotate freely in the horizontal plane. 350-pund lead balls placed nearby were just heavy enough to tug at the small ones, causing the torsion-bar balance to swing slightly (relative to when they were further away). The apparatus was inside a closed room and observed with telescopes to guard against wind draughts, vibrations and other noise. Knowing the masses and distances of all the balls, the force of gravity could be calculated. From that it was possible to derive the mass – weight – of the Earth itself.

Young's light-interference experiment

Young was another one to take a beam of light as a starting point for a simple yet fundamental observation. Using a pinhole in a shutter and a mirror to get a thin beam, he split it by placing a piece of card edgewise in the beam. The result on the opposite wall was a series of alternating white and dark bands – definitely proving that light behaves like waves.

Einstein's explanation of the photoelectric effect, for which he got Nobel Prize, proves equally definitely that light also behaves like particles. We have to call on quantum theory to explain this duality, which was observed using a beam of electrons in an experiment like Young's, by Claus Jönsson of Tübingen.

Computer Science

To conform to the modern definition of a natural science, Computer Science should have both a theoretical basis, more or less formal explanations of phenomena and the way they interact, and empirical evidence strengthening or falsifying observable consequence of such explanations and theories.

Simple reasoning leads us to expect that an "ideal" science would have at least one set of empirical observations for each proposed theory, either strengthening or weakening it. The opposite would indicate a "surplus" of theories that may or may not be falsifiable, but that have not actually been subjected to scrutiny by empiricists.

A study done in 1995 clearly shows that the Software Engineering discipline at least falls far short of this ideal[8], and indeed also falls short of other, comparable natural sciences. An ongoing survey of the leading academic journals shows that the articles describing controlled experiments make up less than 5% of the total volume (10 year average), though with an encouraging rise towards 10-15% in the last year[9].

On the industrial side, it is well known that the amount of hype surrounding methods for conducting software development projects is often anything but scientific, and we do not have sufficient empirical evidence even for many of the accepted truths.

Computer Science is not about programming?

Many scientists seem to be of the opinion that Computer Science is not about programming. As an example, we can look at the Computer Science department at the University of Texas, and their "Fundamentals of Computer Science" course[10]. In the second and third slides of the handouts we learn that there are many disciplines in Computer Science:

- Algorithms
- Formal methods
- Theoretical Computer Science
- Networks
- Artificial Intelligence
- Architecture
- Software Engineering
- Programming Languages
- Systems
  - Operating Systems, Compilers, Databases, Real-Time Systems
- Graphics
- And many, many more (But not programming!)

We are further told that programming is just a skill, admittedly a difficult one to master, but not a science.

From the standpoint of most of the rest of humanity, Computer Science results ultimately must show up as improved was of programming computers, if they are to have any practical significance. However, a famous quotation (attributed to Dijkstra) states that "Computer Science is no more about computers than Astronomy is about telescopes". It is often repeated and used as a justification for pursuing "academic" as opposed to "practical" courses of research.

A personal example is from the time I was hunting for a PhD supervisor, and one professor told me in no uncertain terms that my research should be such that any practical use lay at least 10 years into the future. If it didn't, it wasn't scientific enough – "a PhD is not a programming project".

Going back to the astronomy quotation, an interesting point is that cutting-edge astronomical research has always been intimately tied to telescopes and observing techniques, and those telescopes have been designed by astronomers themselves. Designing, operating or at least understanding the observational tools is essential to any astronomer.

How many Computer Science researchers have an equivalent understanding of what a real-life development project is about, what technology it uses, the properties of available tools, and the skill to use them? I suspect that in their rush to distance themselves from the dirty little details, this understanding has suffered.

One might argue that the insistence on maintaining this dichotomy is one of the reasons why practitioners of the "skill" of programming have so little contact with practitioners of the "science" of Computer Science. Computer Science – and certainly Software Engineering – does not exist in order to reveal fundamental truths about nature itself, in the ways of physics or chemistry. It is a child of technology, and is concerned with the use of technology to practical ends.

However, if we accept the current insistence on Computer Science being distinct from mere programming, it means we must also apply to Computer Science all the criteria of a "real science" – we cannot use the practicalities of programming as excuses.

Theoretical branches

The borderline between abstract mathematics and formal methods in Computer Science is something of a grey area, but there is no doubt that formal methods form a sig-

nificant part of Computer Science. The design and proof of algorithms is one such part. Another is the design and study of modelling languages and relations between them, such as the transformation of Petri Nets or state machines into computer code.

In these branches of Computer Science, we use the same kinds of reasoning as in mathematics, and seek to *prove* results within rigorously defined formal systems.

One example is to define a grammar for a language, and prove that a certain transformation of a valid statement will result in a valid statement in a given, different grammar. Such results stand in their own right, though their implementation or practical use may be of varying difficulty.

Hybrid branches

A discipline like Networks has few problems conforming to the demands and standards of science. It has a firm theoretical basis, from which one can derive models and predictions. The predictions can be tested, either by simulation or by physically constructing a network and loading it with the appropriate kinds of traffic. One can the objectively and precisely measure interesting parameters, such as the time it takes for a certain block of data to go from one place to another, the chance of data loss or distortion, formation of queues in high-load conditions, or other metrics.

Some events, such as collisions of simultaneously transmitted data packets, may be hard to reconstruct individually, but here we can resort to well-established statistical methods and probability calculations.

Most importantly, it is quite practical to conduct *experiments*, both in laboratory settings with artificial, controlled traffic, and studies in real-life situations. It is possible to log the exact content, arrival time etc., of all traffic in a network, and reproduce it with variations at some time later on. Such experiments and reproductions are not hampered by physical location, or distance in time or culture – TCP/IP is the same everywhere, and so is a millisecond.

From real-world observations and experiments, we can then build simulators that permit eve more experimentation. While they are most useful within the boundaries of their design, simulators make it possible to quickly test the properties of new software and hardware without actually having to build all of it.

Software Engineering

There is a considerable contrast when we move to that part of Computer Science that deals with development methods – Software Engineering, the central subject of this essay. Software Engineering spans a wide range of topics and extends into the territories of other sciences – psychology, perception, organization and economics (the list is not exhaustive).

Probably the most central feature of Software Engineering is that it deals with *complex systems*. When I started my career in computers, a programmable calculator with 35 steps and a dozen variables was the only tool. Later computers such as the Apple II, forerunner of the modern PC, had a 64k (65536 characters) memory. This was a system that a human could understand more or less *in toto*, and programs did their essential calculations with a minimum of overhead. Even so, the user interface took up more space than the actual calculations in my programs from that time.

By contrast, modern software runs into millions of lines (Microsoft Windows® is variously estimated around 10 million) of program text. Almost regardless of the way we try to structure and subdivide such behemoths, we cannot keep track of both the big picture and any significant amount of detail simultaneously.

The requirements that software has to satisfy are also increasing almost daily. Initially, a *calculation, search* or *sort* were typically the core of the program, but today a piece of software generally just shuffles and reformats data – calculations are the exception. Instead, there is a lot of communication between program parts, and endless demands for "independence" – it should run independent of what kind of computer, where it is in the world, what kind of network connection it has, what kind of user interface, database, etc. ad infinitum. Even our design tools face the same demands, with some projects requiring design models to be independent of the tool used to make them (and sometimes even the notation!).

Simply fetching data from a database and displaying it has become a task involving a dozen standards, twice that many libraries of prewritten code, and hundreds of thousands of lines.

Software Engineering has the unenviable task of enabling humans and human organisations to handle this complexity. We need to be able to formulate requirements, understand what can and cannot be done, estimate the amount of human, machine and time resources needed to develop a system, design and build it, and verify that it performs with no more than acceptable errors. Clearly this is a task of major proportions.

Why is this important?

In our technical society, we have come to the point that we cannot do *anything* without interacting, directly or indirectly, with computers and software. The extent to which this is the case may not be familiar to nonspecialists, so here is a lengthy example:

In the morning, my digital alarm clock wakes me with its beeping. I open my eyes, and turn on the light, using power that comes in via a computer-controlled international grid of power stations and transmission lines. Even before I woke up, the computer-controlled heating panels turned on to warm the apartment, and in the car the engine-block and interior heaters had started their work.

In the kitchen, another digital clock is part of the timer mechanism on my microwave oven. I turn on the radio, which not only has a computerised station memory, but also uses software internally to decode the radio signals.

My breakfast cereal packet carries a number of bar codes – some read by the store computer when I bought it, others containing information on production lot and content. The built-in microchip in the refrigerator knows when it is time to defrost and performs its task without bothering me. After breakfast, my electric toothbrush is fully charged but not overcharged (that would shorten battery life, so there is a tiny computer taking care of that). It gently warns me after two minutes that I should just brush my teeth, not wear them out.

If I decide to go by bicycle, its computer keeps track of speed, distance, average speed and such trivia; not to mention the industrial robots that built it or the computer-controlled drilling rig in the coal mine that supplied the coal needed for its manufac-

ture. Even the rear light on the bicycle has a tiny microprocessor that blinks the three lamps in a nice pattern, and warns me when the batteries are running low.

Should I instead choose to go by car, starting it is a huge software project, involving at least 5 separate computers and 4 networks (that's inside the car, not during its design!). And if the ABS braking computer fails it might kill me, so there is a lot of redundancy and paranoia in that system.

The keycard lock that lets me in at work is computer-controlled, and so is the milling machine that cut the brass key to my office door.

If I use any kind of money it is tracked by multiple computers and networks. When I pick up the phone, I'm as often as not talking to a computer, using another set of computers to carry the signal and do the switching.

I only got to work, yet I have already interacted with at least a dozen separate computers and pieces of software, and my daily wellbeing is dependent on hundreds more.

Does it matter if all this software is well written, by ordinary, average programmers? It does.

Aside: The purpose of Computers and Software

What are computers for, and why do we pursue them with such vigour? Until fairly recently, technology was an extension of our *mechanical* and *physical* attributes. Humans are not very good by most physical measures when compared to other animals: we cannot run very fast, we need food quite often, we do not adapt all that well to extremes of temperature, etc.

So we used our brains and invented all kinds of technological tools to circumvent these limits, from fur clothes to ladders. Today we can move ourselves in many ways and at breathtaking speeds; we use clothes to survive extreme cold; air conditioning against heat. Trucks, digging machines, micromanipulators and nanotechnology extend our reach into the large and the small. By landing on the moon, we have demonstrated our ability to survive in extremely hostile and inaccessible places.

In contrast, the purpose of computers is to extend our *brains*, the way other technology extends our *bodies*. Can't do a complex calculation? Is it too hard to sort through 100 million names, looking for just one? Don't remember your wedding anniversary? Let the computer handle it!

Condensed history of computing

It started out simple enough with calculating machines[11]. Pythagoras invented the abacus, a device for easing arithmetic. The slide rule dates from the 1600's, and is based on the fact that logarithms shift operations – addition of logarithms (done by sliding two rulers against each other) is equivalent to multiplication of the base numbers.

1642 saw a mechanical calculator for addition and subtraction (Pascal), and in 1673 Leibniz added multiplication and division. Punch cards were used to direct an automated loom by Jacquard in 1804 (based on Vaucansons cards from around 1740), and Herman Hollerith founded the Tabulating Machine Company in 1896.

In the meantime, Charles Babbage made his Difference Engine prototype in 1832, and in 1833 designed the first programmable computer, the unfinished Analytical Engine. Lord Kelvin went the other way, making an analogue Tide Predictor using a cunning

set of differently-shaped wheels to represent the factors, and a string as the integrating mechanism (a beautiful model is on display in Portsmouth).

As ever, war is a great catalyst for technical advance. On the German side, Enigma was a partially-programmable electromechanical coding/decoding device, and it took the Colossus, a 2400-vacuum valve electronic computer to break it in 1943. In the US, ENIAC used 18000 valves to calculate artillery and missile trajectories in 1946. Punch cards by the trainload kept the Holocaust running smoothly[12] in one of the darkest parts of the history of computing, but also aided the Allied side – technology is a willing servant to any master.

From there, development has been exponential, both in methods and hardware. At a recent lecture on numerical methods, it was noted that hardware has become 10^6 times faster since ca. 1960, while the algorithms have become 10^9 times faster during the same interval, giving a total speedup factor of 10^{15} (that's a lot).

Complexity as a feature

This exponential growth in speed and capacity is the enabling factor for our use of computers in new fields. Once usable only for calculations, we now mostly use them for manipulating and shifting huge amounts of data, whether text, sound or images. The development of a ubiquitous and cheap global network has added another important dimension, that of connectivity.

With all this comes a huge increase in the complexity of both the problems we aim to tackle, and the software and hardware needed to do it. Human brains have definite limits to how much complexity (such as the number of factors simultaneously under consideration) they can tackle, so we use *abstractions* in order to hide complexity. The essence of abstractions is to remove (abstract) non-essentials, thereby achieving simplification.

It seems, however, that we are ever balancing on the edge of the possible. As soon as one level of complexity has been brought under control, another is added and the game goes on. Any kind of cutting-edge development work is almost by definition dealing with software of a complexity which is only just possible to keep under control.

From time to time, there are spectacular failures that make the headlines, but for the most part *software works*. It is always ironic to hear people denounce software, after travelling by air to the conference: few activities in modern life are so totally dependent on computer software as air travel.

In spite of the workability of most software, we still feel dissatisfied with the state of the art. We would like to reliably solve even more complex problems, and we would like to do so with a predictable consumption of people, time and other resources.

And the answer is…

The end of computer evolution is by no means in sight, or even on the horizon. Present silicon fabrication principles will run out of steam when they meet physical limits, but the theory and practice of optical and quantum computing are advancing in a promising manner. A speed & capacity improvement by another factor of 10^{12} – 10^{15} should be possible over time, together with global, wireless high-speed connections.

What for? That can only be answered by those alive at the time, and it might be us. But the human appetite for extending the brain seems insatiable, and to a large extent is its own driver. Designing modern electronics without help from massive computers is simply not possible, while increasing computing power keeps opening new applications – for which it is never quite sufficient. The ultimate quantum computer is nature itself, and to what level it can be usefully simulated and manipulated by something smaller is one of the deeper questions.

Empirical Software Engineering

Many sciences have distinctions between theoretical and empirical branches, such as in physics, chemistry or psychology. Common to most is some degree of complementarity, i.e., theories derived from an established basis are tested in experiments, that then cause the theories to be updated, and give rise to new experiments..

The concept of empirical studies in software engineering is nothing new, but as cited[8], there seems to be a large deficit. Most of the empirical studies are *case studies*, in which the researcher either follows or "post-humously" evaluates one or more projects, trying to identify significant factors.

Due to the current lack of a unifying theoretical background, Software Engineering today is a relatively fragmented discipline with many small-scale theories or prescriptions. In most other sciences, that would have given rise to large numbers of experiments and observations to collect data (biology in Darwin's time was a classical example), as people sought to gather the material needed to start building theories. But in Software Engineering, that does not seem to have happened.

There are probably many reasons why this is so, both political, financial, scientific and cultural.

Politics and ambition

Much of the popular literature on software engineering – what you will find in most bookstores, on Amazon etc., is dominated by prescriptive books. The acknowledged gurus in the field (viewed from the industrial standpoint, not necessarily the academic one) are generally people who have *invented* software engineering methods.

Many of them have forceful personalities, which I can attest by personal experience. What they say seems so obviously right that you cannot help but believe them, at least during the lecture. Their copious real-life examples show all the cases where their method worked and brought about success in difficult projects.

Undertaking research that goes against the prevailing fashion is never easy. Standard arguments against conducting experiments are that either they will only confirm the obvious, or they are for some reason not applicable to real life and therefore useless anyway.

Status is gained by inventing a method, not by testing other people's methods. So anyone wishing to become a guru generally looks to improve or invent some new method, not spend years designing and analysing experiments that are then written off in a few sentences ("the design is flawed") by one of the big names.

Heads of research departments are generally also aware of this. Since individual prestige reflects on the organisation, there is not only personal pressure but also institu-

tional pressure to choose a path that leads to recognition – and that generally does not favour the empiricist, as I have shown.

Funding

Experiments are *much* more expensive than case studies. In a case study, it is only the researcher's time that needs to be paid, plus travel and other incidental expenses. Conducting an experiment requires payment for the participants as well as capital investment into workstations, software licenses and infrastructure. The difference can easily be an order of magnitude.

Compared to other branches of science, we are still talking about petty cash. An experiment with 100 participants, paid for 3 days at €100 / hour would cost €240.000. Add 100 PC's with software and a few servers, and we have about €400.000. In medicine, chemistry, physics or almost any other science this is a very small amount (the "ongoing support" list of just one professor in Biostatistics at Johns Hopkins had €3.6 million for studies). Yet experiments of this size are virtually unknown in Software Engineering.

Researchers generally argue that funding is not available. But one reason is that they haven't asked! – there is simply no tradition of large experiments with paid participants, and so it is not usual to ask for funding for it. As a consequence, funding agencies are not used to such requests and are unsure how to evaluate them.

Researcher's qualifications

The experimenter should ideally be aware of relevant parts of psychology, sociology and to a certain extent medicine (where a large amount of experimentation on humans is done). Knowledge of fairly advanced statistical methods for analysing both experimental design and inference power, and the results from experiments, also helps. Finally, there must be sophisticated knowledge of the field of Software Engineering itself, preferably with some industrial experience to help define interesting problems.

A recent article from a group of statisticians lists 36 guidelines for experimenters, ranging from "Apply appropriate quality control procedures to verify your results" to "Present quantitative results as well as significance levels" [13].

This is a rather advanced set of qualifications, and it takes some time to reach a high level of proficiency. Tough requirements are nothing new or unique, but one of the problems of empirical Software Engineering is that (at least until the last 5-10 years) there was little experience to build on, and not much support. It was hard to justify the effort relative to the return in knowledge and status

Experiments on humans

Experiments in Software Engineering are necessarily experiments on humans. We cannot use tapeworms, rats or even chimps as substitutes. We also cannot use Simulators. But experiments on humans are difficult from any standpoint.

First, there is the question of experimental methodology. As researchers, we need to formulate problems and hypotheses that are sufficiently clear-cut to be the subject of experiments. We need to define and isolate the factors influencing the problem under study, and we need to define measurements. Generally, those measurements will be indirect – it is not possible to directly measure *comprehension* of a design method, or

the *efficiency* of a programming model. We have to measure things like time used, the error rate, the amount and quality of code produced, and similar, indirect measures.

There is extensive literature on experimental methodology[14], and it is a complex field. Internal and External validity are the key concepts in designing or evaluating an experiment, and they have special meaning and problems in our context.

Ethics

Experiments on humans always raise ethical issues. Because professional participants are expensive, students have been used in many experiments. This raises the issue of voluntary participation and informed consent. If participation is rewarded with credits, it may not be quite voluntary any more; the same applies to a researcher who "suggests" that his own students in a certain class should participate – there will always be a nagging doubt about the grades of those who refuse, at least in their own minds.

More subtle is the effect of participating. If the participants learn and apply a certain design method during the experiment, those who do not participate miss it. Is the added knowledge then fair payment for participation, and what is the position of those who did not participate?

Similarly, if half the participants in an experiment form a control group that does not learn and apply the method under investigation, they are left with participation but no "reward". If they are paid, those who do not participate are in a similar situation, receiving neither knowledge nor payment, and possibly running the risk of ill will from the professor.

These problems are nothing new, and are well known especially in the medical field. Here one sometimes has control groups who have real disease but receive only placebo treatment. Experiments in psychology sometimes also have the potential to cause harm to participants. Concerns over ethical issues have led to the establishment of a code of ethics in the US[15], and whole books have been devoted to the subject[16].

We can therefore say that though the problems are real, solutions are generally available and accessible to the thoughtful researcher. In the example above, one could ensure that the "control" group in the experiment is taught a different design method, together with the non-participants. In this way all groups would gain roughly comparable amounts of knowledge.

On the issue of participation itself, a researcher should not experiment on his own class, but that of another. Participation would be kept secret, thereby minimising the chance of influencing grades. If this is not possible, one might swap roles by hiring someone from the outside to set the class grades.

Internal validity

Internal validity is the extent to which the experiment actually measures the variable we want to measure, and this variable is influenced solely by circumstances under our control or at least observation. It is obviously not possible to assume total control of all variables in an experiment – variations between individual participants are one example – so we try to minimise, characterise or compensate (in that order) any such influences.

The dependent variable

As mentioned above, we cannot directly measure the most interesting variables, such as comprehension, goodness of design, or even quality. Therefore, one of the fundamental problems is to find the connection between what *can* be measured and what we would *like* to measure. This is called the *dependent variable,* since it depends on the influence of the experimental circumstances. I will use my own replication of an experiment as an example[17].

The experiment attempts to measure the good or bad influence of a certain way of designing computer software (so-called "Design Patterns"[18]), compared to another, "traditional" way. Subjects are given a set of programs and two tasks to perform with each program, typically adding or extending the functions of the program.

All the programs used exist in two versions, one using Design Patterns and one Alternate. Subjects see only one version of each program, and by dividing the subjects into groups and permuting the order and type of program versions, we try to get comparable measurements.

What we actually measure is partly quantitative (the time used) and partly qualitative (the quality of the solution, as judged by an expert). We then assume that better quality or shorter time is a good indication of better comprehension, and assign a positive value to this. This is the transition from a measurable variable to an inferred one, and the subject of Internal validity.

The independent variables

Another subject is the control or suppression of disturbing factors. Ideally, in the above example the *only* differences are between programs and between program versions, and these are the *independent variables* that the experimenter manipulates to obtain measurements. But all the participants are individuals with different skills and backgrounds; they have individual working styles, and they have individual comprehension of the assigned tasks.

The preliminary statistical analysis showed that individual variations were the second largest factor, after differences between the programs maintained, and greater than the differences between Pattern and Alternate versions. So the noise was actually stronger than the signal, to take terminology from signal processing.

This is not a hopeless situation, but requires a fairly sophisticated statistical model to sort out[19], and also enough raw data. Our 44 participants were just enough to get some good data out of it, but 100 or even 200 would have been better.

However, this is a good illustration of the balancing act needed – 200 participants would have been unthinkable with this particular experimental design, which required all of them to work simultaneously, in one place. The logistics were bad enough for 44, and out of reach for 200; also the price would go through the roof.

To get a better answer to the question being investigated, we probably need to use a different experimental design – one that permits us to use more participants, perhaps tries to answer fewer questions, and uses different technological solutions.

External validity

External validity is the extent to which the experiment says something interesting about the rest of the world, outside the lab. Perfect design for internal validity does not ensure external validity.

Traditional problems include the size and complexity of tasks, the technical environment, selection of the participants and the experimental situation in itself.

Size and complexity

Software development projects are usually from a few man-months up to hundreds of man-years. An experiment is considered large if it spans more than a day. The implications for external validity are obvious – it can be hard to infer about year-long projects on the basis of one day.

While difficult, this is not a complete barrier to successful experiments. One can argue that even long-running projects must necessarily be broken into smaller tasks, and an experiment resembles such small subtasks to a reasonable degree. This of course excludes experiments on large-scale design, but it is much better than nothing.

Technical environment

Technical environment is also a challenge. A fully set-up workstation for a sophisticated software developer typically runs a suite of tools that costs considerably more than the computer itself. The installation and customisation of the tools can take days, and the needs of a particular project also take some time to satisfy.

During an experiment there is no time for any of this – all must be set up in advance by the experimenter. There must be enough workstations, and they must be identically configured, otherwise we have a new source of variability and hence noise in the data.

Most experiments have actually sidestepped this challenge and used a pen-and-paper method instead. The programs and tasks are printed on paper, and participants make their comments and changes using pens; computers are not involved. Again, this poses problems for external validity. *Details matter* when you make programs, and some seemingly trivial detail of the programming language or environment may suddenly cause you to spend much more time than expected. Also, when programming on paper, you cannot actually test your solution to see if it compiles (is syntactically correct) and works.

There have been some cases of pen-and-paper experiments being replicated in a real programming environment, and though the actual time used increased, it seems that overall quality was not affected. Thus, it seems that our expectation that a programming environment encourages participants to test their solution, and thereby achieve higher final quality, is not borne out. Conversely, pen-and-paper experiments are not automatically invalid.

Participants

Selection of participants is another problem. In order to perform a valid selection, we must first define the *population* we wish to study (or make inferences about), and then choose how to sample it. Is the population all programmers? Just consultants? In a particular language or technology? What about the application domain – banking, engineering, process control, administrative?

Second, how do we sample? Given that we are performing an experiment requiring active participation, it is hard to use classical random sampling – some of the selected participants may not turn up. On the other hand, we should be extremely vary of self-selecting samples (i.e., volunteers), since these tend not to be representative. It is easy to imagine a consulting company sending only out-of-work employees to participate in an experiment, and one would reasonably expect the most able people to be fully booked, so this would result in a skewed sample.

Since professional participants must be paid for their time, many experiments have used students instead. Arguments have been raised both for and against, and the main issue is whether students are representative of the programmer population in general. Those results that we do have from comparative studies suggest that Computer Science students actually represent the upper part of the general population, which agrees with what one would expect from the simple fact that many programmers do *not* have higher education in the field.

Various methods exist for countering these problems. Simply paying well is one important step; this increases the willingness of participants who are otherwise busy to take the time to participate. Pre-screening, preferably with a programming task relevant to the experiment, can be used to eliminate those who are not qualified to take part, and to gain some insight into the proposed sample. Careful, conscious definition of the population to be studied, followed by selection of organisations to participate, will ensure that we at least understand what we are studying and where the experimental data can be used to make inferences.

Experimental stress

Being in an experiment is much more like an exam than the everyday work situation for most people. Even though there are often tight deadlines and lots of overtime and pizza in software development, an experiment is different.

The classical classroom setting, where 25 participants sit hunched over their keyboards, occasionally sneaking a look at the guy in the next place to see how far he has come with the task, is obviously conducive to stress. In such a situation, it may well be that the experiment is measuring the individuals' ability to work under intense stress more than anything else.

There are several ways this can be managed. Depending on the experimental design, it may be possible to run the experiment at the participants' own workplace, using the same technical equipment that they use every day, and in the same offices. This obviously poses logistical and technical problems (we have to ensure working configurations, reasonably identical working conditions, etc.), but those can be surmounted.

If the experiment allows it, we can also make sure that people sitting next to each other do not work on the same task at the same time, thereby reducing the comparison to other people and resulting stress. If that is not possible, we can try to make people start in staggered intervals.

If the experiment has to be done in a lab, it can be turned into smaller cubicles using temporary walls, and participants seated back-to-back so they do not see each other.

Payment can be arranged for a certain minimum time (thereby encouraging people to work fast, since they will not lose money by finishing early), and also for time spent after that (so people who are slower do not give up).

And still it turns!

From all the objections it is easy to conclude that experiments in Software Engineering are either useless, irrelevant or both. The amount of planning that needs to go into an experiment is huge, and issues of many kinds must be considered. The researcher must take into account ethics, software engineering, psychology, experimental design, statistics, logistics, finance, politics and technology, and be reasonably proficient in all of them.

There is a real risk that prestigious journals will reject submissions on the grounds that results are not statistically significant with a P-value < 0.001 (or some other arbitrary limit), or that "we knew this anyway". And if the experiment *fails* to find significant results in a situation where they are expected, this is often taken as an indication that the experiment itself was deficient, and not that the (for instance) method being investigated is actually bogus and has no real effect. Having spent a year and a lot of money, this is a demoralizing outlook.

However, I believe we *must* continue to experiment. If our results are inconclusive, we must improve our methods. If we can conclusively show that some programming method has no measurable effect, that is a result that must be taken seriously.

The Journal of Universal Computer Science has taken a step by establishing a "Forum for Negative Results". This is meant for experiments or other observations that are methodologically sound, but fail to find the observed result.

Computer Science, and in particular Software Engineering, has been denounced many times over for being neither a "real" science, nor engineering. Given the phenomenal success of the scientific method in other areas, and the unquestionable importance of Software Engineering, making it more like the other natural sciences should be a priority. And as I hope to have shown, this *cannot* be done without solid empirical work. Yes, it is difficult, but it is one of the foundations on which Science rests. Without it, much of the rest will be in vain.

References

1. Eriksen, T. B. **Filosofi og vitenskap i antikken** (Universitetsforlaget, Oslo, 1983).
2. O'Connor, J. J. & Robertson, E. F. (School of Mathematics and Statistics, University of St Andrews, Scotland, 1999) Source: http://www-history.mcs.st-andrews.ac.uk/history/References/Pythagoras.html.
3. Tranøy, K. E. **Filosofi og vitenskap i middelalderen** (Universitetsforlaget, Oslo, 1983).
4. Fløistad, G. **Filosofi og vitenskap fra renessansen til vår egen tid** (Universitetsforlaget, Oslo, 1983).
5. Chaitin, G. J. **Gödel's Theorem and Information**. *International Journal of Theoretical Physics* **22**, 941-954 (1982).
6. Popper, K. **Objective Knowledge** (Clarendon Press, 1966).
7. Crease, R. P. (PhysicsWeb, 2002) Source: http://physicsweb.org/article/world/15/9/2.
8. Tichy, W. F., Lukowicz, P., Prechelt, L. & Heinz, E. A. **Experimental Evaluation in Computer Science: A Quantitative Study**. *Journal of Systems and Software* **28**, 9-18 (1995).
9. Sjøberg, D. in **Survey of Controlled Experiments in Software Engineering** (Oslo, 2002).
10. Scott, M. (VIT, 2002) Source: http://www.cs.utexas.edu/users/scottm/cs307/Handouts/lec1-4up.pdf.
11. Kogge, P. M. in **History of Computing, from CSE322: Computer Architecture II** (2000).
12. Black, E. **IBM and the Holocaust** (Time Warner Paperback, 2002).
13. Kitchenham, B. A. et al. **Preliminary guidelines for empirical research in software engineering**. *IEEE Transactions on Software Engineering* (2001).
14. Christensen, L. B. **Experimental Methodology** (Allyn & Bacon, Boston, 2001).
15. **Ethical Principles in the Conduct of Research with Human Participants** (American Psychological Association, Washington, D.C., 1982).
16. Kimmel, A. J. **Ethical Issues in Behavioural Research** (Blackwell Publishers, Inc., Cambridge, MA., 1996).
17. Prechelt, L., Unger, B., Tichy, W. F., Brössler, P. & Votta., L. G. **A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.** (2002).
18. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. **Design Patterns: Elements of reusable object-oriented software** (Addison-Wesley, Reading, MA, 1995, 1995).
19. Diggle, P., Liang, K. & Zeger, S. **The analysis of Longitudinal Data** (Oxford University Press, Oxford, 1994).