Epistemic Opacity, Confirmation Holism and Technical Debt: Computer Simulation in the Light of Empirical Software Engineering

Julian Newman

Epistemic Opacity, Confirmation Holism and Technical Debt: Computer Simulation in the Light of Empirical Software Engineering

Julian Newman

Birkbeck College, University of London, UK

juliannewman@ymail.com

Abstract. Software-intensive Science, and in particular the method of modelling large and complex systems by means of computer simulation, presents acute dilemmas of epistemic trust. Some authors have contended that simulations are essentially epistemically opaque vis and vis a human agent, others that complex simulation models suffer from an inescapable confirmation holism. We argue that the shortcomings lie in the failure of modellers to adopt sound Software Engineering practices, and that the elevation of computational models into superior epistemic authorities runs counter to principles that are common to both Science and Software Engineering.

Keywords. Simulation Models · Surveyability · Holism · Epistemic Trust

1 Introduction

Software-intensive Science, and in particular the method of modelling large and complex systems by means of computer simulation, presents dilemmas of epistemic trust in acute form. In general, epistemic trust may be placed in colleagues, informants, methods, practices and instruments. Such trust may appear inconsistent both with the organised selective scepticism which is at the core of the scientific attitude and with the provisional nature of corroboration – yet the necessary levels of specialisation in modern sciences, and the consequent division of labour in laboratory teams and research programs, conspire to make some level of trust inevitable.

Instances of serious scientific malpractice or error do sometimes spark a moral panic, leading journal editors to adjust publication requirements – for example regarding data deposition or description of authors' contributions and acceptance of responsibility regarding the paper as a whole – but the need to trust fellow scientists, both within the team and externally, seems impossible to eradicate. Conversely, the acceptance of particular experimental results as sound tends to enhance trust not only in the researchers who produce them, but in the methods, practices and instruments employed: the "Matthew effect" [46] leads to a "credibility cycle" in the accumulation of idiosyncrasy

credits, professional authority and funding [14,15], [36,37]. It is within this context that we should critically assess claims that scientific practices are self-vindicating (e.g. [23], [63]).

A number of authors have recently explored the relationships between experimentation, simulation and human cognitive limits. This has led some, we shall argue, into a misapplication of epistemic trust. For example, Symons & Horner [56] have argued that the defining characteristic of Software-intensive Science is that it is impossible within any realistic timescale to test the correctness of all possible paths through the software on which such a science depends, so that the error distribution in Software-intensive Science is unknown – nevertheless Symons remains wedded to the view that "people should trust" models [5], [55]. Similarly, Humphreys [25,26] has argued from the (allegedly essential) epistemic opacity of much or most Computational Science, including computer simulations, to the need for a "post-anthropocentric epistemology". Winsberg [63] suggests that the techniques used to construct simulation models are self-vindicating in the manner predicated by Hacking [23] of experiments and instruments, and – writing with Lenhard [38] – that complex system models have to be accepted or rejected as a whole, in a novel form of confirmation holism. Frisch [18, p 177], extends the concept of epistemic opacity to cover not merely the inability of the cognitive agent to know all the epistemically relevant elements of a process at a given time, but also this impossibility of identifying the contribution of a model's different components to its overall performance. We shall therefore refer to the general position espoused by Humphreys, Lenhard, Winsberg and Frisch as "The Epistemic Opacity Doctrine".

1.1 Structure of the Paper

Our argument proceeds as follows: First, we give a brief account of the positions that we intend to question; next, we advance two arguments critical of these positions: we shall call these the *argument from the theory of the instrument* and the *argument from scientific norms*. Taken together these arguments lead to the conclusion that the acceptance of computer simulations as superior epistemic authorities is neither desirable nor necessary.

The first argument starts from the position that Empirical Software Engineering, rather than Computer Science, is the essential "theory of the instrument" upon which to ground the ability of computer simulations to warrant scientific knowledge claims; we sketch the concepts and practices of "Engineering", "Engineering Sciences", "Software Engineering" and "Empirical Software Engineering"; examples from Empirical Software Engineering are then used to explore the Ontology and Epistemology of software, and we suggest that findings from empirical studies of other large software systems evolved over a long time period are particularly pertinent to understanding the epistemic issues that arise with respect to complex simulation models. Of particular interest are findings concerning the relation between software architecture and the incidence and persistence of defects, and concerning the phenomenon of "technical debt".

The second argument, from scientific norms, builds on studies of the practices whereby scientific communities manage the issues surrounding epistemic trust. Trust amongst scientific colleagues is not indiscriminate: epistemic dependency is socially managed by means of specific strategies of indirect assessment, leading to considered judgements of the degree of trust to be placed in another scientist, and the areas in which that colleague can be regarded as reliable. These social practices, which allow trust relations to be managed without abandoning local scepticism, cannot be extended to an instrument regardless of its computational power.

Both arguments lead to rejection of the claim that opacity in computer simulations justifies regarding them as superior epistemic authorities.

2 Software

2.1 Opaque Models as Superior Epistemic Authorities

The output of a simulation model does not, prima facie, appear to have an objective status comparable with data captured by observation or experiment using defined reproducible procedures. Counter to this Winsberg, Humphreys and others emphasise parallels between experiment and simulation in practices which are said to "carry with them their own credentials" [25,26], [63]. By essential epistemic opacity, relative to a cognitive agent, Humphreys means that it is impossible for that cognitive agent, given his or her characteristics, to know all of the epistemically relevant elements of a computational process. Humphreys views essential epistemic opacity as reflecting the limitations not of the simulation method itself but of the human agent, and thus as evidence for a "non-anthropocentric epistemology" recognising computational tools as a superior epistemic authority.

The possibility of testing a highly parameterised simulation model via the hypothetico-deductive method can indeed be open to doubt; moreover empirical measurements are often not available on the scale needed to evaluate model outputs. Even were appropriate data available, Lenhard & Winsberg [38] argue that climate simulation models face epistemological challenges associated with a novel kind of "confirmation holism": it is impossible to locate the sources of the failure of any complex simulation to match known data, so that it must stand or fall as a whole. This is because of three interrelated characteristics which they regard as intrinsic to the practice of complex systems modelling – "fuzzy modularity", "kludging" and "generative entrenchment".

In "fuzzy modularity", different modules simulating different parts of the complex system are in continual interaction, thus it is difficult to define clean interfaces between the components of the model. A "kludge" is an inelegant, 'botched together' piece of program, very complex, unprincipled in its design, ill-understood, hard to prove complete or sound and therefore having unknown limitations, and hard to maintain or extend. "Generative entrenchment" refers to the historical inheritance of hard-to-alter features from predecessor models. The critic of the Epistemic Opacity Doctrine must

confront three possibilities. Confirmation holism may be essential to and unavoidable in complex systems modelling, or embedded in specific disciplinary practices of Climate Science, or may exemplify a failure to observe, recognise and apply available and well-established sound Software Engineering practices in simulation software projects. [1]

Belief in the essential epistemic opacity of Computational Science points to the frst alternative but we shall argue that the third better characterises this phenomenon. It should be noted that attempts to develop large complex software systems beyond the technical and project-management capabilities of those concerned is a recurrent problem well known to software engineering consultants and academics, and that it is thought to be particularly prevalent in publicly-funded organisations. Several writers have identified factors that work against the use of good Software Engineering practices in scientific computing, including the tendency to manage projects according to scientific goals rather than software quality goals, the variety of specific contexts in which scientific software is developed and deployed (which is a potential obstacle to learning from other Computational Science projects), the long lifetime of much scientific software leading to the accumulation of legacy code and the obsolescence of computational techniques used, and various communicational, organisational and resource factors that predispose scientists to write their own code rather than to employ a professional software engineer to do so [12], [29],[50].

2.2 Theory of Software-based Instruments

A computer simulation model is an instrument whose functionality is delivered through software. Since our knowledge of computers is, to an overwhelming extent, knowledge of the behaviour, affordances[2] and malfunctions of software, getting the epistemology of software right is an essential precondition for any correct and informed philosophical approach to other epistemological issues in which computers are implicated. Thus the "theory of the instrument" in computer simulation studies is the theory of software.

The 'internal' perspective on software adopted by its designers and programmers is not our principal or most reliable source of knowledge concerning its actual nature and probable behaviour, although empirical investigation can show us, amongst other things,

[1] The use of rigorous Software Engineering methods in Computational Science has been promoted by a number of recent initiatives, for example the Software Sustainability Institute http://software.ac.uk/ and the Karlskrona Manifesto [3].

[2] An affordance is a perceptual feature of an artefact with which a user can interact in order to evoke a behaviour: a familiar example of an affordance is an "icon" on which the user can click in order to open a file or start an application. The concept derives from Gibson's psychology of perception [19,20],[48].

how to maximise the 'surveyability'[3] of a software artefact thus increasing the chances that an internal view will correctly anticipate the artefact's behaviour.[4]

Software as an Immaterial Artefact. We now present an account of software as an immaterial artefact, produced and maintained through the practice of Software Engineering. The perspectives from which we treat the matter are those of Software Engineering practice and Empirical Software Engineering evidence, and thus consciously distinct from the Computer Science approach that informs Turner's analysis of computing artefacts [60,61]. The conceptual analyses of miscomputation and malfunction, related to Levels of Abstraction, presented by Fresco, Primiero and Floridi [10,16,17] come closer to a Software Engineering approach, but do not address Empirical Software Engineering as our main source of scientific knowledge of the factors determining the reliability of software.

An Engineering practice involves the systematic, knowledge-based solution of engineering problems, where an engineering problem is understood to be one whose full solution is the creation, validation and maintenance of a functional artefact. Whereas the goals of science must be characterised in cognitive terms, and the goals of engineering are characterised in terms of the delivery of functionality subject to constraints. Scientific knowledge may assist the engineer in achieving these goals, but is not for the engineer an objective in its own right.

Let us now consider three important points about software: it is an immaterial artefact, but not on that account incomplete; like all functional artefacts software is ontologically dual, having an objective structure designed to realize an intentional function; full knowledge of its objective structure cannot be obtained a priori.

Some have characterised computer programs as a new kind of mathematical object[5] – yet the complete software product is much more than the algorithm(s) that it implements. An ontology of artefacts should be aligned upon the schemes of individuation[6] familiar to the creators and users of those artefacts. An artefact is often assumed to be a physical object whose structure is designed to serve a function [33, p xxv]. Software is then described as an "incomplete artefact" needing a suitable machine

[3] Turkle [58] draws attention to the oddity of the established usage, in computing fields, whereby "transparency" refers to the hiding of details rather than making them apparent. She goes so far as to say that in what she refers to as "a culture of simulation" … "transparency means epistemic opacity". To avoid confusion on this point, we prefer to use 'surveyability' rather than 'transparency' as the contrast-term for 'opacity'.

[4] The "internal" and "external" perspectives mentioned here should not be confused with the theories of the "inner" and "external" environment of an artefact discussed below.

[5] According to Lamport [35] "Floyd and Hoare … taught us that a program is a mathematical object and we could think about it using logic." Mathur [43, p. 36] treats this as a widely accepted orthodoxy ("It is often stated that programs are mathematical objects"), but points out that the complexity of such an object and of its environment form obstacles to effective proofs of correctness of large-scale software products. See also Turner [59].

[6] The concept of "scheme of individuation" is drawn from Situation Theory (e.g. [9]).

on which to run [45, p 90]). But computing hardware serves only the most generic function.

The specific functionality in which users are interested lies in the application software that is running above a platform constituted by hardware and layers of system software. Thus for the software engineer, as for the user, applications software stands itself as a complete artefact, while a suitable platform of machine and systems software is an important element of its intended *operating environment*. The fact that an artefact depends upon a particular operating environment does not make it incomplete. Recognition that an artefact may be immaterial brings the ontology closer to the schemes of individuation prevalent in a society where such artefacts are pervasive.

Notwithstanding the above, software participates in the ontological duality characteristic of all artefacts: an artefact has an objective structure and an intentional function [33]. Yet the objective structure of software is not a physical mechanism. Developing this type of immaterial artefact is an engineering practice. Discovering the characteristics that determine or limit the capacity of software's objective structure to deliver intentional functionality goes beyond engineering practice and requires the notion of "Engineering Science".

Empirical Software Engineering. Following Boon [4], we distinguish between Engineering on the one hand and, on the other, "Engineering Sciences" which are scientific investigations motivated by the need to support Engineering practice. Given the immaterial nature of software artefacts, we relax Boon's stipulation that Engineering Sciences use the same methods as the Natural Sciences. An engineering science appropriate to supporting Software Engineering has to borrow methods from a range of sciences better adapted to deal with immaterial artefacts: these are, broadly, the behavioural, organisational and statistical sciences.

But why should Software Engineering not be accepted as a science in its own right? After all, the software engineer employs various theories and models in constructing solutions to problems, and subjects his or her solutions to rigorous tests which, it can be said, are closely analogous to experimental tests of theory in the Natural Sciences. This is, for example, the position advanced by Northover et al. [49] who state that "the susceptibility of software to testing demonstrates its falsifiability and thus the scientific nature of software development." They further argue that software developers "are responsible for establishing, by careful a priori reasoning, an overall 'theory' that guides the development of working software programs."

There is an unfortunate ambiguity here, which we may clarify by reference to Simon's distinction between the Inner and External environments[7] of an artefact, and to the idea of Claims Analysis developed within Human-Computer Interaction studies [7],[54]. In Simon's analysis, the "Inner environment" of an artefact is the technology which the designer uses to produce the behaviour at the interface which delivers the function of the artefact, while the "External environment" is the intended operating

[7] The apparently paradoxical idea of an "Inner Environment" derives ultimately from Cannon's "Wisdom of the Body" [6].

environment from which the functional requirements derive [54]. According to Claims Analysis, the design of an artefact, in particular its interface, involves implicit claims about the External environment in which the artefact is to be deployed (often including claims about the capacities of its intended users [7]). Reinterpreting Simon into the language of Kroes & Meijers, these claims about the External environment are a theory which provides a context for identifying the intentional functions of a proposed artefact. On the other hand, a theory of how the artefact can be made to produce certain behaviours – a theory of its Inner environment – describes the structure that delivers the intentional function (in the case of software, as noted above, this structure is not material: it may, arguably, be considered as a logical structure, although its actual manifest behaviour may not conform to expectations arrived at through logical analysis).

Northover et al's "overall theory that guides the development of working software programs" may therefore refer to a theory of the Inner environment – what structures will produce the desired behaviour at the interface? – or it may be a theory of the External environment – into what human activities or distributed information environment must the artefact fit? A test scheme in the context of developing a specific software artefact is not an attempt to disconfirm or corroborate either type of "overall theory": the theory of the External environment stands rather as a quality control standard. The requirements from which a test case derives are a model of the External Environment which is not invalidated if the software fails the test [2, p 429]. [8]

The parallel between testing software and testing a theory will not, therefore, bear the weight of the Northover argument that "each test case … is like a 'scientific experiment' which attempts to falsify part of the developer's overall theory". The software under test is an artefact in the making: it is not an experimental, quasi-experimental or controlled observational setup designed for the purpose of testing either the theory of the Internal or that of the External environment. Putting either type of theory on a sounder basis requires a differently designed activity with different goals. Hence we conclude that just as materially-based Engineering practice has need of Engineering Sciences, so does Software Engineering need its own engineering science; it is the need for such a science, oriented to but distinct from Software Engineering practice, that has given rise to "Empirical Software Engineering". Two important caveats must however be entered regarding our rejection of the Northover thesis. First, the authors correctly locate systematic testing as a feature common to Science and Software Engineering. Second, "reflective practice" is characteristic of Engineering as a whole and can be an important source of theoretical ideas. Our critique of the limits of the "internal view" of software does not deny that useful theories of Software Engineering can arise out of practitioners' experiences in software development and testing; rather it emphasises that such theories need to be tested by "Empirical Software Engineering".[9]

[8] A system failure may also occur because the requirements specification fails to capture correctly a critical feature of the External environment [16,17].

[9] Cf Popper on hypotheses and theories [51].

Actual practices of Software Engineering are amongst the objects that can be studied and evaluated by Empirical Software Engineering. So also are the characteristics, both designed and emergent, of tools used in Software Engineering practice (e.g. programming languages, modelling tools, process models and standards, test plans, configuration management tools [13], etc.), and the ways that Software Engineers, individually and in teams, actually interact with those tools and with one another. Whereas Software Engineering as a specialism within Computer Science originated in the late 1960s [21,22], Empirical Software Engineering arose in the 1990s. Two landmark events in 1996 were the commencement of the EASE[10] conferences and the creation of a new journal, *Empirical Software Engineering*. Its founding editors defined its scope as "the study of software related artifacts for the purpose of characterization, understanding, evaluation, prediction, control, management or improvement through qualitative or quantitative analysis. The quantitative studies may range from controlled experimentation to case studies" [24]. They stated that current 'mainline' Software Engineering journals did not "adequately emphasize the empirical aspects of Software Engineering," and described Software Engineering as "not currently … a 'fact-based discipline' " and as one of the few technical fields where practitioners seldom required (nor even particularly desired) "proof in the form of well-developed, repeatable trials before accepting and acting on claims."

Regarding the relationship between Computer Science and Software Engineering, Harrison & Basili discern a general lack of appreciation for empirical work within Computer Science. The favoured research paradigm in Computer Science, they write, "tends not to follow *the scientific method where one establishes a hypothesis, conducts an experiment or otherwise collects data and then does a statistical analysis to substantiate or reject the hypothesis.* Rather, Computer Scientists tend to get rewarded for building systems and doing some sort of generalized analysis of the performance, benefits, etc. … We can see that this bias … propagates itself into the way most software engineers are trained …". [24, *emphasis added*]. By 1999 the long-established *IEEE Transactions on Software Engineering* had responded to the new trend with a Special Section on Empirical Software Engineering [28]. Writing in another IEEE journal, Zelkowitz & Wallace [66] criticised the misuse of the term "Experimentation" in the Computer Science community to describe a weak implementation example or 'proof of concept' and lacking rigorous evidence: they refer to such experiments as 'Assertions'.

From the foregoing, certain characteristics of Software Engineering and Empirical Software Engineering may be discerned:

1. The so-called "Mertonian norm" of organised local scepticism is a value commitment that is common to Software Engineering practice and to Science (including Empirical Software Engineering as a branch of Science). This is not contradicted by our preceding arguments against the view that Software Engineering is in itself a science. The norm of (local) scepticism is explicitly invoked in the critique of "Assertions" [66].

[10]Variously interpreted as "Empirical Assessment in Software Engineering" or as "Evaluation and Assessment in Software Engineering".

2. Our knowledge of the actual characteristics of software is empirical, not a priori, despite the role of mathematical knowledge (such as "computational templates" [21]) in motivating and guiding the creation of software artefacts.
3. In Software Engineering practice, a theory serves as a standard of correctness against which the artefact is judged; but in "Empirical Software Engineering", as it is a branch of Engineering Science, theories do stand to be judged by the results of empirical tests.
4. The statement that software developers "are responsible for establishing, by careful a priori reasoning, an overall 'theory' that guides the development of working software programs" [49] is true of computational templates, but not of the complete software artefact, of which the behaviour cannot be known a priori, hence the practice of systematic testing of software, and the growth of Empirical Software Engineering as an attempt to bring empirical evidence to bear upon knowledge claims and practices in the Software Engineering field.
5. As a corollary, while Mathematics can show us in some cases what software cannot in principle do, Mathematics cannot give us certainty about what software actually will do.

Methods. The range of methods used in Empirical Software Engineering includes controlled experimentation, project monitoring, studies of legacy data, case studies, field studies and systematic reviews which incorporate any or all of the foregoing in an attempt to support evidence-based Software Engineering Practice [12],[27],[32],[34],[39,40],[42],[64].

Controlled experiments in Empirical Software Engineering [57] generally involve work with human subjects under varying conditions in which independent variables reflect factors of interest to Software Engineering practice. Subjects for such experiments are often students of computing subjects but may on occasion be professional software engineers. Large scale software projects require teamwork, therefore experiments in which factors are manipulated at the level of the individual subject may lack ecological validity. This has motivated some studies, oriented to the effectiveness of team processes, which are effectively quasi-experiments and not true controlled experiments, independent variables such as personality attributes or team climate being measured but not strictly manipulated. Appropriate and realistic experimental materials are required for such experiments, and these are often derived from published code, most commonly found on Open Source development sites. Concerns have been expressed about whether those materials are truly representative of Software Engineering in general [64]. One solution is to develop suitable infrastructure for hosting a repository of code from real-world projects, with capabilities to support experimenters in creating and replicating experimental studies. Do et al [11] describe such an infrastructure dedicated to supporting controlled experimentation upon different software testing techniques.

In other methods (Project Monitoring, Legacy Data Studies, Case Studies and Field Studies) data generated from real-world activities are used not as experimental materials but as actual observations. Whereas Case Studies in social sciences typically involve collecting qualitative data from human participants, in Empirical Software Engineering

a Case Study can involve the extraction of quantitative data from project records created over a lengthy period, potentially giving insights into the nature of the software artefact as it develops and changes, and into the characteristics of software that create the most difficult cognitive problems for the practising software engineer. Such records include fault reports, change requests, versioning and defect tracking [11],[52]. Comparative studies using such data (e.g. multi-case studies) may however face validity threats from organisational or team variations in reporting and recording practices.

Empirical Characteristics of Software Defects. A useful example of empirically-based theory development through statistical analysis of a long term case study of project records is the work of Li et al. [41], which can serve as an example of the way in which characteristics of software that could not be known a priori are revealed by empirical investigation. A software architecture captures basic design decisions which address such issues as performance, reliability, security, maintainability and interoperation. This study concerned the relationship between software architecture, location of defects, and the difficulty of detecting and correcting those defects.

Architectural decisions made early in the development process address qualities central to system success. As many as 20% of the defects in a large system can be attributed to architectural decisions, and these defects can involve twice as much effort to correct as defects arising from mistakes in requirements specification or in the implementation of software components [39],[65]. Thus at least for a large-scale software system the theory of the Internal Environment may assume considerably greater importance than that of the External Environment: the developers may have greater difficulty in ensuring that the artefact delivers its functionality, than in determining what that functionality should comprise.

Li et al [41] point out that architectural decisions typically affect multiple interacting software components, and as a result architectural defects typically span more than one component: they therefore concentrated on the problems of finding and correcting "multiple-component defects" (MCDs). To this end, they conducted a case study based on the defect records of a large commercial software system which had gone through six releases over a period of 17 years. Compared to single component defects they found that MCDs required more than 20 times as many changes to correct, and that an MCD was 6 to 8 times more likely to persist from one release to another. They identified "architectural hotspots" consisting of 20% of software components in which 80% of MCDs were concentrated, and these architectural hotspots tended to persist over multiple system releases. This study provides an excellent example of the part played by the "Engineering Science" of Empirical Software Engineering in developing a relevant body of theory upon which practice in Software Engineering can build.

Returning to the "Epistemic Opacity" theme, the lessons of Li's and similar research on the aetiology and persistence of software defects are highly relevant to interpreting Lenhard & Winsberg's account of the factors that make it impossible to trace the reasons why particular outputs from a complex systems model fail to match observed data [38]. As summarised in section 2.1 above, they argue that complex simulation models in general, and climate models in particular, are—due to fuzzy modularity, kludging, and generative entrenchment—the products of their contingent respective histories and that

climate models are as a consequence analytically impenetrable in the sense that it has been found impossible to attribute the various sources of their successes and failures to their internal modelling assumptions. They suggest that complex models in general exhibit a form of confirmation holism, but nevertheless claim that the failure of climate models to converge is a good sign: "It would even be reason to be suspicious if science would announce a coherent and unanimous result about this topic." We would argue, rather, that disagreement because one has built instruments whose function one does not understand scarcely constitutes a healthy scientific pluralism, that fuzzy modularity implies a failure to define a clear architecture for the simulation software, and that a kludge is a software defect waiting to manifest itself.

Alexander & Easterbrook [1] comment that there are very few representations of the high-level design of global climate models on which to base discussion, planning and evaluation; while they were able to create top level architectural diagrams of eight climate models by means of code mining, we argue that the fact need for such representations of the architecture to be discovered, rather than being created and maintained by the developers, confirms a failure of the latter to engage with crucial factors that would determine the quality and surveyability of their simulation software. Nevertheless, when Pipitone & Easterbrook [50] compared defect density in three climate models with that in three open-source projects they found that on this measure of quality the climate models scored well. They discuss a large range of validity threats that may undermine this apparently favourable result for the climate modellers [50, pp 1017-1020], including possible differences in defect recording practices and in what they describe as "successful disregard" of certain types of defect.

Technical Debt. A further line of research in Empirical Software Engineering concerns the consequences of making early programming decisions on a purely pragmatic basis (e.g. in order to get the system working) – in other words, the consequences of kludging. It has been shown that such short cuts create "Technical Debt" [8],[52] on which interest will accrue in the form of error and maintenance costs throughout the lifecycle of the software product. Technical Debt, first noted in 1992 [8], [53], became a major research focus in Empirical Software Engineering around 2010. Kruchten et al [34] summarise current views of Technical Debt in terms of visibility/invisibility and in terms of maintainability and evolvability. Visible elements include new functionalities that need to be added and known defects that need to be fixed, but in their view "what is really a debt" is the invisible result of past decisions that negatively affect the future value of the software artefact. Ways in which this invisible debt can burden the developers and stakeholders include architectural problems giving rise to the hard-to-correct type of multi-component defect discussed above [41], associated shortcomings in documentation, and factors making existing program code difficult to understand and modify, such as code complexity and violations of coding style. Studies reviewed in [34] present evidence that visible negative features depend to an important degree on less visible architectural aspects (see also [30, 31]). The version of epistemic opacity described by Lenhard & Winsberg and by Frisch manifests many of the characteristics of Technical Debt [18],[38]. Pipitone & Easterbrook [50], in their discussion of the apparently low defect density of the climate models they studied, write that climate

modellers may have learned to live with a lower standard of code and development processes, and that a "net result may be that [they] incur higher levels of Technical Debt". Note that this assumes that Technical Debt is defined in terms of "problems in the code that do not affect correctness, but which make the code harder to work with over time," whereas the central point of Lenhard & Winsberg concerns the loss of the ability to "tease apart the various sources of success and failure of a simulation and to attribute them to particular model assumptions of different models" [38, p 253].

Moreover, the treatment of Technical Debt by Kruchten et al shows that the concept should not be restricted to maintainability issues, and studies such as that by Li et al [41] demonstrate the impact of software architecture upon the incidence and persistence of defects. As software is an immaterial human artefact, the objective structure which delivers its functionality depends upon surveyability, which implies an adaptation to human cognitive capacities. Technical Debt is, then, ultimately a phenomenon of relative opacity (i.e. lack of surveyability) in an artefact. Since the effectiveness of techniques for enhancing surveyability can be warranted by Empirical Software Engineering research, this opacity is contingent, not essential.

2.3 Implications of Empirical Software Engineering for the Epistemic Opacity Doctrine

Firstly, where a large software system is epistemically opaque with respect to a human agent, this opacity is not an essential characteristic arising from its size but is contingent upon development practices and in particular upon architectural design. Decomposition into manageable, surveyable components is an essential architectural strategy for managing complexity. Failure to perform such decomposition adequately at the design stage will certainly make the software itself epistemically opaque, but it would be perverse to regard this as endowing the resultant artefact with superior authority or the capacity to carry its own credentials. In Software Engineering practice, defects are expected: human activity is error-prone. Yet well architected software is not epistemically opaque: its modular structure will facilitate reduction of initial errors, recognition and correction of those errors that are perpetrated, and later systematic integration of new software components. Nothing intrinsic to complex simulation modelling prevents the application of these principles, but kludging in the early stages of model building will create "Technical Debt" which will be charged in the form of *contingent* epistemic opacity and its consequences. Simulation software is epistemically opaque (when it is) not because of the inability of a human agent to check through every possible execution path from beginning to end, but because of a failure of model builders to adopt the practices which are known to promote surveyability and effective error management. The "generative entrenchment" identified by Lenhard & Winsberg can act as a barrier to a clear, clean architecture for a model built out of previous models. For example, the failure of many global climate models to respect conservation of energy is thought to result from previous ocean and atmospheric models having different grid scales and different coastal representations. Ad-hoc fitting together of pre-existing models that have not been designed to be components of a global model creates problems

that have to be fixed at the stage of model tuning, and the model tuning itself introduces further opacity into the behaviour of the overall model [18],[44]. We have also seen that institutionalised local scepticism is characteristic of Software Engineering practice, as well as of Science. Credulity towards simulation software as a superior epistemic authority, which must be accepted as a whole, runs entirely counter to this norm. This theme is further developed in section 3 below.

3 Managing the Limits of Epistemic Trust

A simulation model must be understood as a tool which can play a part, along with other resources, in a scientific argument; such an argument depends upon human judgement which, fallible though it may be, cannot legitimately be replaced by an allegedly superior epistemic authority. The argument from the essential epistemic opacity of Computational Science to a non-anthropocentric epistemology runs counter to best practice in Software Engineering and to empirical results of Software Engineering Science. In this respect it is self-defeating.

Humphreys' argument for 'non-anthropocentric' epistemology amounts to considering the result of a computation to be a warranted knowledge claim, even though a human user cannot readily trace the dynamic relationship between the initial and final states of "the core simulation". Humphreys provides a valuable and painstaking analysis of the process of building and justifying computational models. In particular his discussions of computational templates and correction sets in [25] deserve closer attention by practitioners. The weakness of his treatment, however, is that it ignores the Software Engineering process that creates the actual simulation model, effectively inviting us to abdicate human critical judgment in the face of superior computational capacities.

Wagenknecht [62] has argued that the ubiquity of trust, in research as in daily life, does not imply that trust is indiscriminate and blind: rather epistemic trust amongst scientific colleagues has inherent limitations. On the basis of an ethnographic study, she shows that this epistemic dependency is managed by means of specific strategies of indirect assessment, including dialogue practices and the probing of explanatory responsiveness. From the foregoing, one could conclude that post-anthropocentric epistemology is impossible, since it would require us to manage trust in a cognitive agent that is not itself a social actor. Although Wagenknecht also describes scientists as resorting to "impersonal trust", she uses this term to describe trust in the epistemic quality of scientific communities and institutions, gatekeeping and peer review mechanisms, not in inanimate instruments and artefacts.

4 Conclusions

We have not resolved the dilemmas of epistemic trust, nor have we aimed to do so. What we have established, however, is that the surveyability of a computer model does not depend on the ability of a human agent to perform its calculations in real time. The

practice turn in Philosophy of Science should not degenerate into a form of credulity towards the instruments of Computational Science, particularly since the traditional scientific norm of local (moderate) scepticism is also a core commitment of Software Engineering, a discipline which, we have argued, provides the "theory of the instrument" for Computational Science. "Generative entrenchment" of features from predecessor models exemplifies a widespread problem of legacy code that manifests itself in a wide range of long-lived software systems in many fields other than Science; the narrative of "Technical Debt" provides a rationale within which the Software Engineering community is developing strategies and techniques for addressing these problems, and which we argue may be fruitful for understanding the resistance of climate models to analytic understanding.

The slide from the firm ground of practice into credulity towards opaque computations has undoubtedly been exacerbated by failure of journal editors to require authors to publish or deposit their code [47]. Of greater importance, though, is a philosophical re-evaluation of the nature of software as a human artefact and of the contribution of the different and distinct computing disciplines. The present article has concentrated upon critique, but the required philosophical work should not neglect the positive aspects of the research programme within which Lenhard & Winsberg's "Holism" paper [38] is embedded, a programme which emphasises the interactive nature of much Computational Science, summarised as "a pragmatic mode of scientific understanding" or "methodology for a virtual world" [63], which is "Gibsonian" in the sense of placing emphasis on active exploration as the basis of perception [20]. A future task will be to explore how that perspective can exploit the Gibsonian merits of Human-Computer Interaction [7],[9],[48] without losing the grasp of detail and surveyability that characterises the research programme of Empirical Software Engineering.

Acknowledgements: The author is indebted to Giuseppe Primiero and to an anonymous reviewer, for comments on previous drafts of this paper.

References

1. Alexander, K., Easterbrook, S.M.: The software architecture of climate models. Geosci. Model Dev., 8, 1221-1232 (2015)
2. Angius, N.: The Problem of Justification of Empirical Hypotheses in Software Testing. Philos. Technol. 27, 423–439 (2014)
3. Becker C., Chitchyan R., Duboc L., Easterbrook S., Penzenstadler B., Seyff N., Venters C.C.: Sustainability design and software: The Karlskrona manifesto. Proc. 37th Int. Conf. Software Engineering, Volume 2 pp. 467-476. IEEE Press (2015)
4. Boon, M.: In Defense of Engineering Sciences: On the Epistemological Relations Between Science and Technology. Techné 15:1, 49-70. (2011)
5. Boschetti F., Fulton E.A., Bradbury, R.H., Symons, J.: What is a model, why people don't trust them, and why they should. In Negotiating our future: Living scenarios for Australia to 2050 107-119. (2012)
6. Cannon, W.B.: The Wisdom of the Body. Norton, New York, NY (1932)
7. Carroll J.M. (ed.): HCI models, theories, and frameworks. Morgan Kaufman (2003)

8. Cunningham, W.: The WyCash portfolio management system. Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications, Vancouver, British Columbia, Canada, *A19, ACM (1992).
9. Devlin, K.: Logic and Information. Cambridge University Press (1991)
10. Dewhurst, J.: Mechanistic Miscomputation: A Reply to Fresco and Primiero. Philos. Technol. 27: 495-498 (2014)
11. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering, 10, 405-435 (2005)
12. Easterbrook S.M., Johns, T.C.: Engineering the software for understanding climate change. Computing in Science & Engineering. 11(6) 64-74 (2009)
13. Estublier, J., Leblang, D., Hoek, A.V.D., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management. ACM Transactions on Software Engineering and Methodology (TOSEM), 14(4), 383-430 (2005)
14. Fleck, J.: Informal information flow and the nature of expertise in financial services. Int. J. Technology Management, 11(1-2), 104-128 (1996)
15. Fleck, J.: Contingent knowledge and technology development. Technology Analysis & Strategic Management, 9(4), 383-398 (1997)
16. Floridi, L., Fresco, N., Primiero, G.: On malfunctioning software. Synthèse 192, 1199-1220 (2015)
17. Fresco, N., Primiero, G.: Miscomputation. Philos. Technol. 26, 253-272 (2013)
18. Frisch, M.: Predictivism and old evidence: a critical look at climate model tuning. Euro. Jnl. Phil. Sci. 5, 171-190 (2015)
19. Gibson, J.J.: The Theory of Affordances. In R Shaw & J Bransford (eds) Perceiving, Acting, and Knowing. Erlbaum (1977)
20. Gibson, J.J.: The Ecological Approach to Visual Perception. Houghton Mifflin (1979; Republished 2014 Psychology Press and Routledge Classic Editions)
21. Grier, D.A.: Software Engineering: History. In Encyclopedia of Software Engineering, pp. 1119 – 1126. Taylor & Francis (2011)
22. Grier, D.A.: Walter Shewhart and the Philosophical Foundations of Software Engineering. Third International Conference on the History and Philosophy of Computing (HaPoC 2015) Pisa, Italy, 8-11 Oct 2015. (2015)
23. Hacking, I.: Representing and Intervening. Cambridge University Press (1983)
24. Harrison, W., Basili, V.: Editorial. Empirical Software Eng. 1(1), 5-10 (1996)
25. Humphreys, P.: Extending Ourselves: Computational Science, Empiricism and Scientific Method. Cambridge University Press. (2004)
26. Humphreys, P.: The Philosophical Novelty of Computer Simulation Methods. Synthèse, 169, 615-626 (2009)
27. Jedlitschka, A., & Pfahl, D.: Reporting guidelines for controlled experiments in software engineering. In International Symposium on Empirical Software Engineering, 2005. IEEE, pp. 95-104. (2005)
28. Jeffery, D.R., Votta, L G. (eds): Special Section on Empirical Software Engineering, IEEE Trans. Software Eng..25(4) (1999)

29. Kanewala, U., Bieman, J.M.: Testing scientific software: a systematic literature review. Information and Software Technology 56, 1219-1232 (2014)
30. Kazman R., Cai Y., Mo R., Feng Q., Xiao L., Haziyev S., Fedak V., Shapochka A.: A case study in locating the architectural roots of technical debt. Proc. 37th Int. Conf. Software Engineering, Volume 2 pp. 179-188. IEEE Press (2015)
31. Kitchenham, B., Pfleeger, S.L.: Software quality: the elusive target. IEEE Software, 13, 12-21 (1996)
32. Kitchenham, B., Pfleeger, S.L., Pickard, L. M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering, 28(8), 721-734 (2002)
33. Kroes, P. & Meijers, A. (eds.): The Empirical Turn in the Philosophy of Technology. Emerald (2000 [republished 2009])
34. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Software, 29(6), 18-21 (2012).
35. Lamport, L.: The Future of Computing: Logic or Biology. Talk, Christian Albrechts Univ. Kiel, 11th June 2003. Downloadable from http://research.microsoft.com/en-us/um/people/lamport/pubs/future-of-computing.pdf (2003)
36. Latour, B.: Insiders out. Ch 4 in Latour, B, Science in Action: How to Follow Scientists and Engineers through Society. Open University Press, pp. 145-176 (1987)
37. Latour, B. & Woolgar, S.: The cycle of credibility. In: Barnes, B. & Edge, D. (Eds), Science in Context: Readings in the Sociology of Science. Open University Press, 35-43 (1982)
38. Lenhard, J., Winsberg, E.: Holism, entrenchment, and the future of climate model pluralism. Studies in History and Philosophy of Modern Physics 41 253–262 (2010)
39. Leszak, M., Perry, D.E., Stoll, D.: A Case Study in Root Cause Defect Analysis. Proc ICSE'00 22nd International Conference on Software Engineering. Limerick, Ireland pp. 428-437. IEEE (2000)
40. Lethbridge, T., Sim, S., Singer, J.: Studying Software Engineers: Data Collection Techniques for Software Field Studies. Empirical Software Engineering, 10, 31-341 (2005)
41. Li, Z., Madhavji, N.H., Murtaza, S.S., Gittens, M., Miranskyy, A.V., Godwin, D., Cialini, E.: Characteristics of Multiple-component Defects and Architectural Hotspots: A large system case study Empirical Software Engineering 16: 667-702. (2011)
42. MacDonell, S., Shepperd, M., Kitchenham, B., Mendes, E.: How reliable are systematic reviews in empirical software engineering? IEEE Transactions on Software Engineering, 36(5), 676-687 (2010).
43. Mathur, A.P.: Foundations of Software Testing. Pearson (2008)
44. Mauritsen, T., Stevens, B., Roeckner, E., Crueger, T., Esch, M., Giorgetta, M., Haak, H., Jungclaus, J., Klocke, D., Matei, D., Mikolajewicz, U., Notz, D., Pincus, R., Schmidt, H., Tomassini, L.: Tuning the climate of a global model. J. Adv. Model. Earth Syst., 4, M00A01, doi:10.1029/2012MS000154.

45. Meijers, A.: The Relational Ontology of Technical Artifacts. In Kroes, P., Meijers, A. (eds.) The Empirical Turn in the Philosophy of Technology. Emerald, pp. 81-96 (2000 [republished 2009])
46. Merton, R.: The Matthew effect in Science. Science, 159(3810) 56-63 (1968)
47. Morin, A., Urban, J., Adams, P.D., Foster, I., Sali, A., Baker, D., Sliz, P.: Shining Light into Black Boxes. Science, 336, 159 – 160 (2012)
48. Norman, D.: The Design of Everyday Things. Basic Books, New York. (1988; Original hardback title: The Psychology of Everyday Things; 2nd ed. 2002)
49. Northover, M., Kourie, D.G., Boake, A., Gruner, S., Northover, A.: Towards a Philosophy of Software Development. J. General Philosophy of Science. (2008)
50. Pipitone, J., Easterbrook, S.: Assessing climate model software quality: a defect density analysis of three models. Geosci. Model Dev., 5(4) 1009-1022 (2012)
51. Popper, K: Conjectures and Refutations: The Growth of Scientific Knowledge. 2nd ed. (revised). Routledge, London (1965)
52. Runeson, P,, Host, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, 14(2) 131-164 (2008)
53. Schmid, K.: Technical Debt – From Metaphor to Engineering Guidance. IEEE Transactions on Software Engineering 25(4) 573-583 (1999)
54. Simon, H.A.: The Sciences of the Artificial 3rd ed. MIT Press (1996)
55. Symons, J., Boschetti, F.: How computational models predict the behavior of complex systems. Foundations of Science, 18(4), 809-821 (2013)
56. Symons, J., Horner, J.: Software intensive science. Philos. & Tech., 27(3), 461-477 (2014)
57. Tichy, W.F.: Should computer scientists experiment more? IEEE Computer, (5), 32-40 (1998)
58. Turkle, S.: The Fellowship of the Microchip: Global Technologies as evocative objects. In Suarez-Orozco, M.M., Qui-Hilliard, B.D. (eds.) Globalised Culture and Education in the New Millenium. U. California Press, Berkeley, CA, 97-113 (2004)
59. Turner, R.: The Philosophy of Computer Science. Stanford Encyclopedia of Philosophy http://plato.stanford.edu/entries/computer-science/
60. Turner, R.: Computational Artefacts. IACAP Conference 2013 (2013)
61. Turner, R.: Programming Languages as Technical Artifacts. Philosophy & Technology 27:377–397 (2014)
62. Wagenknecht, S.: Facing the incompleteness of epistemic trust: Managing dependence in scientific practice. Social Epistemology, 29(2), 160-184 (2015).
63. Winsberg, E: Science in the Age of Computer Simulation. U. Chicago Press (2010)
64. Wright, H.K., Kim, M., Perry, D.E.: Validity concerns in software engineering research. In Proceedings of the FSE/SDP workshop on Future of Software Engineering Research. ACM pp. 411-414 (2010)
65. Yu, W.D.: A software fault prevention approach in coding and root cause analysis. Bell Labs Technical Journal 3(2), 3-21 (1998)
66. Zelkowitz, M., Wallace, D.: Experimental Models for Validating Computer Technology. IEEE Computer 31(5), 23-31 (1998)