Brittleness and Bureaucracy: Software as a Material for Science

Matt Spencer
Oxford University

Through examining a case study of a major fluids modelling code, this paper charts two key properties of software as a material for building models. Scientific software development is characterized by piecemeal growth, and as a code expands, it begins to manifest frustrating properties that provide an important axis of motivation in the laboratory. The first such feature is a tendency towards brittleness. The second is an accumulation of supporting technologies that sometimes cause scientists to express a frustration with the bureaucracy of highly regulated working practices. Both these features are important conditions for the pursuit of research through simulation.

1. Introduction

Computer simulations play a key role in many domains of contemporary science. The question of how they transform science has received a lot of scholarly attention. Many have concentrated on outlining the epistemological similarities and differences between simulation and experiment and theory (for example Ostrom 1988; Keller 2003; Winsberg 2009) and on the diversity of kinds of modelling and representing found across domains of science (Morrison 1999; Giere 2004). A third direction, which I explore here, looks instead at how working with computers, and working in software environments in particular, gives this kind of computational science research its distinctive character. Following Knuuttila (2005b; 2011), I take modelling to be practical engagement, in which software serves as the material for building and running simulations.

Telling the story of Fluidity, a finite-element code for simulating fluid dynamics, as it grew from a small project to a large software system, I show how piecemeal growth exerts pressures on research through a tendency

This research was supported by the UK Arts and Humanities Research Council.

Perspectives on Science 2015, vol. 23, no. 4
 doi:10.1162/POSC_a_00184

towards software brittleness. The trouble that brittleness entails provided a key axis of motivation for the scientists as the software grew. In an attempt to definitively push the group beyond these difficulties, the Fluidity developers eventually rewrote the code, and adopted a number of new technologies and working practices to bolster their defenses against the re-emergence of such problems in the future. While countering brittleness, however, these steps themselves transformed the working environment, and placed new demands on research time that altered the nature of research, which some saw as a bureaucratic burden. This is a singular story, but these transformations of the nature of research can be expected to reflect the contours of the wider landscape of computational science, most importantly, allowing us to grasp the differences between small and large-scale code development, and the dynamics of transition between the two.

The case study of Fluidity is based on an 18 month period of ethnographic observation and extensive interviews with the members of the Applied Modelling and Computation Group (AMCG) at Imperial College in London.

2. Workability in Research

What makes a good model? A normative question like this has two kinds of answers. The first has gained a lot of attention. A good model is something that plays a role in inference. Usually this means it is a good representation: a computer model, for example, that generates data with a good level of fit with some comparison data set (Oreskes et al. 1994; Kleindorfer et al. 1998; Lahsen 2005; Bailer-Jones 2003). This then provides grounds for inferences about the thing represented, or for inferences about the reliability of the model when simulating systems or problems for which no such comparison data set exists.

A second kind of answer puts the question of representation on one side, and regards a model in terms of interactivity, how it affords more or less efficacious manipulative possibilities to scientists. Tarja Knuuttila has pushed for an appreciation of a model as something that may be judged in its own right, as what she calls an "epistemic artefact" (Knuuttila 2005a; Knuuttila 2011). A model is a certain kind of thing, with its own characteristic constraints and affordances, interactive possibilities it offers to an embodied user (Knuuttila 2011; Gibson 1986). Moving from the subjective meditation of abstract thinking, to what Gaston Bachelard called the "objective meditation" that occurs between the researcher and their equipment (1984, p.171), this view of modelling calls our attention to the material properties of models.

Questions of materiality have been raised by philosophers seeking to understand scientific inferences, focusing, for example, on the way arguments may reference the physicality of computational processes that occur when code is run (Parker 2009), or on comparisons with "same stuff" arguments

about material causes found in other kinds of modelling (Guala 2002). Knuuttila's departure from this tradition lies in the emphasis on modelling as something that happens prior to the formalization of arguments.

The practical dimensions of software have yet to be extensively studied, probably because their concreteness is not as easily appreciated as that of more intuitively physical models (for example, Justi & Gilbert 2003). Hence, many studies of simulation regard computational research to be abstract, symbolic work (Heymann 2006; Sundberg 2008, p.17). However, citing Klein's work on "paper tools" (2002), Knuuttila stresses that "even in the case of symbols and diagrams, the fact that they are materially embodied as written signs on a paper accounts partly for their manipulability" (2011, p.269; see also Goody 1986; Netz 2003). "Without materiality," she claims, "mediation is empty" (2005b, p.1267).

Taking up this theme, Mary Morgan has recently highlighted the "workability" of productive models (Morgan 2012). If we look at code in the making, we can ask how software becomes efficacious as material for thought and action, a medium for exploration and inspiration. We can also ask how, conversely, workability may threaten to break down and compromise research. While scientists are motivated by epistemic goals, those I encountered during my fieldwork were equally motivated by the need to respond to these kinds of threats, to shore up their research systems against them and to make them as efficacious as possible for the future.

An initial clarification is necessary to point us in the right direction. When I first arrived at AMCG to conduct my study of computer modelling, I regarded Fluidity as a model. That was how it was casually talked about, as a versatile model of fluids. I soon realized that this is subtly misleading. If we are interested in the medium of their research, that medium is not a singular model, but a modelling environment. "I think of Fluidity as a code or modelling framework, that is not itself a model," explained NK, a senior scientist. The code can be set up in limitless ways, with a vast array of options, coupled models, sub-models. The concept of a model too easily implies that it is a representation, that it entails a directional relation with some represented target, what Suarez calls "representational force" (2010). A modelling framework, on the other hand, has no neat relationality. It is a software toolkit, a working environment within which models may be created. Workability is therefore not simply a property of a model, but of a constellation of coupled and integrated software systems that embodies a potentiality for creating models, a medium in which any scientist sets up their simulations through the use of existing code, the integration of new code, and through tweaking and refining the many controls on the system, some according to explicit rules, but many more according to the tacit "feel for the game" inculcated by an experienced user

(Bourdieu 1977). While Knuuttila has pointed the way with her concept of "epistemic artefact," the image this provides is too discrete and singular to account for the workability of software modelling frameworks, which exist more as settings, or contexts, for research.

3. Piecemeal Growth and Agility

Fluidity, like much scientific software, is continually evolving, and this perpetual change is an index of ongoing investments of effort and interest by scientists pushing their research in new directions, requiring new or tweaked functionality from the code, or using existing components for new purposes. The piecemeal growth that characterizes this software is a consequence of the manner in which science is funded. It is rare to get funding to build a large software system from scratch. These systems become large through a history of additions and extensions. Within the terms of individually funded research projects it is usually only feasible to build a small software system, tailored to the specific goals of that project. Once built, however, it can serve as a platform or a proof of concept for further projects, which extend it in new directions, adding new functionalities, exploring new possibilities, eventually leading towards funding for increasingly ambitious extensions. Success in producing results thus tends to "grow" scientific software (Basili et al. 2008, p.29).

It has been noted that the piecemeal growth of scientific software aligns its development style with the "agile" approaches popular in some areas of commercial software development (Easterbrook and Johns 2009). While conventional development strategies rely on comprehensive prior specification of the functionalities of the completed product, the signatories to the "Agile Manifesto" promoted a style which characterizes software as something that is open and changing, made in a context of changing requirements (Beck et al. 2001). During a project, the client's priorities may change, the market for the product might change, competitors may bring out rival products; it is not, in many cases, easy to see at the beginning what the product will need to be, so agile approaches focus on embracing changing requirements. Furthermore, success for a software product tends to lead to further modification, to bring in new features and keep it up to date with other systems (Matsumoto 2007, p.478).

An emphasis on keeping software projects agile leads to characterizations of the merits of source code that resonate with ideas of the "workability" of models. With his concept of "habitability," Richard P. Gabriel, for example, challenges the conventional mode of judging software as a finished product. "Software needs to be habitable," he says, "because it always has to change. Software is subject to unpredictable events: Requirements change because the marketplace changes, competitors change, parts

of the design are shown to be wrong by experience, people learn to use the software in ways not anticipated" (Gabriel 1996, p.13). This trend in thinking has drawn attention to source code as an environment within which developers work, something that was previously often hidden behind teleologies of final products or end user experience. "Habitability makes a place livable, like home. And this is what we want in software—that developers feel at home, can place their hands on any item without having to think deeply about where it is" (Gabriel 1996, p.11). While it implies no specifically epistemic goal, habitability nevertheless captures the core of workability, a baseline of manipulability without which epistemic projects would be hard to realize, a technical condition that operates in interaction with the epistemic (Rheinberger 1997, p.32).

No comprehensive specification exists for what exactly it is that Fluidity should end up being. This is common in computational science (Segal and Morris 2008, p.18). Computational scientists are agile workers because their goals are bound up in the development process, rather than in an eventual destination. The research projects at AMCG use simulations built *en route* with an evolving code (Sundberg 2008, p.5). Studies of scientific practice have emphasized the under determination of what it is that scientists aim at (Rheinberger 1997), and it is notable that one of the pioneers of the agile approach later discovered in Andrew Pickering's writings many of the same themes that inspired him to push for new strategies in software development (Marick 2008). If the researcher knows exactly what they are aiming at, it is hardly an inspiring topic for research (see Polanyi 1983). Similarly, where specifications dominate software cultures, it is hard to see the creative side of development practice (Evens 2006). So Fluidity develops according to a meandering, iterative logic, being put to many uses for many different kinds of studies, each which stresses and extends it in new directions. No singular idea exists for where it is going. One senior member of the group told me that the goal was eventually to make it "a whole earth model." Another said that they were aiming at an ultimate "toolkit" for making simulations. A third regarded the software as secondary to the community of expertise they were building through its development and use. All these perspectives feed into the management of Fluidity, but none with the determining character of a final cause.

4. Fluidity's Early Life

Fluidity began its life in the late 1980s as part of a Ph.D. project looking at small scale fluid phenomena for industrial applications. The original author, CK, brought his code to Imperial College a few years later when he joined the Applied Modelling group, which at that time was primarily concerned with studying radiation transport problems for nuclear reactor

engineering. Fluidity was soon integrated into the toolkit of the group, because this opened the door to studying nuclear applications involving fluids, such as those found in fissile solution reactors. With the assistance of a new coupling code, "FETCH," Fluidity was coupled to "EVENT," which was the main AMCG nuclear modelling code.

This emerging interest in fluids within AMCG sparked much further development of Fluidity and in the late 1990s new staff was brought on board to extend the code to new applications such as multiphase flow (where fluids transition between liquid and gas phases) and to render it capable of running efficiently on highly parallel supercomputer architectures. At this point, AMCG was composed of 15 scientists, and was roughly evenly divided in its work between radiation problems and fluids problems. In the early 2000s, however, a major new funding initiative began which doubled the size of the group within four years and rapidly accelerated the pace of development of the code. The main part of this project was a major extension of Fluidity from its previous incarnation as an engineering code, to a huge new realm of geophysical applications, from oceans, coasts and rivers to atmospheric and mantle physics.

Fluids became the largest area of study within AMCG, and by 2010, there were around 30 scientists actively developing Fluidity. The engineering and small-scale computational fluid dynamics research continued, while geophysical applications accounted for more than half of what this much bigger development team was studying. The beginnings of this expansion in 2001–2003 saw the commencement of many new geophysical research projects, but it also saw a major shift in the way that the science was done because the expansion of the code had led to threats to its workability.

The expansion of the group was an expansion in the number of people involved, but this also brought with it an expansion in the number of different agendas for which Fluidity was being used. The group experienced new demands for collaboration and communication. At the same time, the increase in scale resulted in an increased rate of change for the Fluidity code base. The software was being changed often, by many people working on many different projects. These stresses placed a burden on the team that was exacerbated by the fact that much of the code had not been written with an expectation of future massive expansion. The priority during the early life of Fluidity had always been to deliver the short-term aims of specific research projects, and there had been little time or money available to make the system maximally extensible. Furthermore, as is the case for many computational science teams, the majority of scientists working on Fluidity had not come from a computer science background. They were predominantly mathematicians, engineers and physical scientists. Even now, few PhD students coming to AMCG have worked on complex

software systems before they arrived. They learn on the job and as KU pointed out, this has good and bad effects: "when you have a team of good programmers around you you learn by osmosis. But when you have a team of bad programmers around you learn by osmosis as well!" When Fluidity was a small/medium sized enterprise, there was a degree of tolerance for idiosyncratic working practices. What mattered was getting good simulations, and the quality of software was less of an issue. But as it grew, this tolerance dwindled.

> As a mathematician I was never interested in coding practices. It didn't bother me. As long as my code ran fast and did what it was supposed to do I was happy. So my codes were often a tremendous mess. Someone like WS would have been none too happy seeing that. But then that is no problem when you are on your own—no-one else has to look at it!" (IM)

> Over the years Fluidity had been adapted … bit by bit for different applications. We ended up with quite an unmanageable mess, what coders call "the code becomes brittle." There were so many hidden assumptions in the code that as soon as you change one detail the whole thing breaks down (HP).

The trouble with the code in the early days of the expansion was expressed in very tangible terms. The code had become "brittle." This term is a commonplace within software cultures. Brittle code is the opposite of robust code (it is important not to confuse the robustness of code with the alternative use of the term as a measure of agreement between the outputs of different models—see Parker 2011). Brittle code breaks more often than robust code. But the fundamental problem with brittle code is not so much that it breaks more often—all code can be expected to break when being developed – it is that when it does break it is hard to figure out exactly why. Then, when you have figured out why it broke, it is hard to fix it. And when you do implement the fix, there is a good chance of causing further problems. Brittleness is a somewhat loose term that captures the experience of coders struggling with working in what has become a very difficult and frustrating medium. Developers have problems tracing causation in brittle technical systems, leading to an expansion of effort required for even relatively small interventions.

> Have you seen the old code? It was written without any levels of abstraction whatsoever. That makes the code very difficult to understand because what you see is a whole load of very low level mathematical operations and then you have to work out for yourself what the big picture is and what all the bits are doing. And that is

> hard to see. It was missing an awful lot of modern software engineering, which is about making code that is either error free by design or at least easy to define what the errors are (WS).

Brittleness compromises habitability. Working with and upon a brittle Fluidity rendered research difficult and time consuming. If a great deal of time is spent fixing bugs and tracing the logic of convoluted sections of source code, there is very little sense of the code as an open and flexible medium. Its affordances dwindle. The modelling environment is less conducive to just "trying things out" in the set-up of a simulation, following a whim or an informal inclination. Not only do these processes take much longer, it is harder for new collaborators to be brought on board. One complaint with the old Fluidity was that it was hard for new Ph.D. students to properly become experts in the code in the limited time span of their doctorate.

> The interface originally was very cumbersome. Only two or three people in the group were able to use it. It was a big text file with lots of random names. They all had to be six letters so it used all sorts of crazy acronyms. It was very hard to modify the option system but they were constantly adding new functionality so basically the numbers became encoded in more and more complicated ways. For example, if you put a minus sign in the time step that might mean something special. Or you put in a very large number with different meanings for the different digits … Nobody really knew why it broke when you changed something. It was a big project to step by step change things to see where it broke and figure out why (HP).

5. Scale and Brittleness

What makes a software system brittle? Fluidity became brittle as it became big and as the group developing it became larger, brittleness being relative to scale. It is, however, difficult to talk rigorously about the size of software. There is no perfect measure. The most common measure is the number of lines of code in the source code repository. This is the software written by humans before it is compiled into binary code that can be executed on hardware. But counting lines of source code is somewhat arbitrary: different software languages use different rules of formatting that take up different amounts of space. This is compounded by the fact that Fluidity involves different languages working together. According to some relatively standard calculations, however, Fluidity gets currently estimated at over a million lines of code. At the point of the rewrite it already comprised more than a quarter of a million.

But the size of the source code is not a good measure of the complexity of the code (Booch 2008). The complexity of the software architecture had become a problem for the Fluidity developers. Because the code had grown in piecemeal fashion, there was very little large-scale planning at the start, and many additions were tacked on in inconsistent ways. A code that is very large in terms of lines of code need not be complex for a human reader to find their way around. It may be very intuitively organized. Conversely, even a relatively small code may have a torturous organization with many headaches in store for a potential navigator.

A third element of the scale of software is the complexity of the processes that it computes. Some software performs tasks that require a lot of space to be encoded, but which are relatively conceptually simple, whereas something like Fluidity is computing a lot of very complex mathematics, and this considerably affects how manageable its source code is.

6. Making Fluidity Robust

As the group's expansion got under way it became clear that the code would only become more brittle as more scientists carried on adding new functionality. Problems were compounded by the rate of development. Some scientists were already refusing to work with the latest version of the code because they were sick of results that they had gotten one week no longer being reproducible a week later. The response was two-fold. Firstly, there would be a complete rewrite of the code, completely reorganizing its architecture and the style of programming in which it had been written. Secondly, a number of supplementary practices and technologies would be adopted to help co-ordinate the work and to change the working style of the group.

> When the grant to develop ICOM ["Imperial College Ocean Model" is Fluidity applied to oceans problems] arrived, all of a sudden a lot of people were working on the development side. Code organization and management became critical. We hired people who had good ideas about testing, code structures, modularization (HP).

The complete rewrite of Fluidity took a small team about a year. While there was no funding earmarked specifically for this task, a number of new projects were just getting off the ground and this work was built into their early phases. The rewrite required a huge amount of work, but its pay-off was to be a newly efficacious, much more robust code. It is an extreme measure to rewrite a code from scratch, but the new code re-implemented the same core algorithms that had been developed over the previous 15 years of Fluidity's lifetime. While the work of expressing these algorithms in

software had to be redone, all the work that it took to devise them in the first place did not need to be repeated.

The "new Fluidity" did what the old Fluidity did, but it expressed it in a more reader-friendly and manipulator-friendly way, organizing the core algorithms so it was much easier to find your way around the source code. The new code used a newer version of Fortran and employed many new stylistic techniques, all the way down to new rules for naming variables. The modularity of the new code allowed new functionality to be added easily and to interface with existing modules in a standardized manner. A new options system with a clean and clear graphical user interface complemented this process, speeding up the process of running new simulations and tweaking them in the course of their investigation.

> Things have moved on dramatically in the last few years … People came in too who knew more about software development and modern programming. Slowly what happened was that good practice and rewriting the code in modern Fortran happened. The code we have now has effectively zero lines in common with the code that I started on. The algorithms are the same—these are the important part, the hard part. The early stages of working on these codes are getting the algorithms. But then get an algorithm and there are hundreds of ways that you can code that, lots of choices of how you implement it: different languages, different structures, different orders of things that you do things in. One of the big things that happens now in dev. [developers'] meetings are debates over this kind of thing because there are lots of ways of doing something: What is the best way to do it? And people have strong feelings about this, making sure that things are useful for other people and future-proof. In the old days you were only worried about getting something working (NK).

The modularity of the new version of Fluidity was taken a long way. Modularity in its extreme means that parts of Fluidity could in theory be transplanted into very different modelling frameworks without having to be retooled for the new job. For example, in the main body of the code

> we don't code anything about the discretisation, so we can let things change without having to recode. The code doesn't even know it is a fluids model. It just knows it is solving PDEs [Partial Differential Equations] (WS).

A complete rewrite of a software system is a radical tactic, and from a naïve standpoint it may appear to signal a damning assessment of the old version of the code. However, it has, in principle, no bearing on the epistemic

legitimacy of the scientific arguments made on the basis of the simulations created by the old code. From a deductivist viewpoint interested in the validity of scientific discourse, brittle and robust codes make little difference. Exactly the same simulation, the same computational process, can be generated by a brittle code as by a robust code. Or if these processes differ, they may differ in no epistemically significant manner. Just because a code is robust does not mean that the simulation it produces will give a more legitimate answer to a given question. This issue is one of validation and verification, which pertain to a finished simulation and not to the process of writing and manipulating software. In other words, brittleness and robustness are properties of an environment for modelling that affect the processes through which simulations are created, but which do not transfer to the resulting simulations.

From a sociology of practice standpoint, however, the robustness of code matters greatly. A robust code embodies greater potential than a brittle code because it causes less frustration and less time absorbed with setting things up and dealing with bugs. It is more habitable and facilitates a greater degree of manipulation, and therefore offers a lot more potentiality to surprise the scientists, to lead him or her down an unexpected path of exploration towards innovative research.

On the other hand, practical considerations are never wholly isolated from the arguments made afterwards by the scientists. In the write-up of results, the justification of claims is accomplished by a drawing together of a patchwork of arguments. Verification and validation are usually the major components, but they are never conclusive. A code that is known to be robust and well-written is likely to include fewer bugs, so less likely to involve bugs affecting the solution that have eluded detection in verification and validation. Even where these issues are not explicitly referred to in publication, research communities are often well aware of the merits and drawbacks of each other's systems. Researchers move between groups, interact at events, and seek out opportunities to try each other's codes, and the judgments they form about each other's systems make a difference to the general appreciation of their claims.

7. Social Technologies

Fluidity's robustness was increased by the re-write. But it would be a mistake to think about manipulability solely as a property of the software itself. Everything depends on working practices. There is no straightforward way to isolate the technology from the wider ecosystem of techniques through which it is brought into use. The overcoming of the brittleness of Fluidity was also the overcoming of the frustrations that arose from the transition to working in a larger group. At the same time as the code was

being rewritten a number of what we could call "social technologies" were introduced to the group, a holistic institutional adaptation, reconfiguring working procedures.

Research at AMCG now takes place between the gears of many different systems for managing the speed of change, the size of the group, and maintaining consistency and coherence of the evolving software. A full version control system tracks every change made to the source code, while an automated testing suite checks every new version of the code for errors.

> Part of our quality control process for the code is all this testing and that is both in order to make sure that new features that people develop don't cause problems but also to make sure that people haven't sort of messed it all up or literally made a spelling mistake or get a wrong syntax (QH).

While I was there, the scientists were experimenting with a code review system, so that every new section of software is reviewed by another developer to try to pick up errors or just inconsistencies of structure or style. A manual and user guide, and set of example simulations, are all kept up to date so that new users can get to grips with the code quickly and easily.

> Some reviews are extensive … [The time commitment] can be expensive but probably not as expensive as a bug getting in (HU).

Since November 2010, Fluidity has been open source, so new users from around the world can download the software and they inevitably need to be provided with support. By expanding the user-base, being open source also has the advantage of enhancing the probability that bugs will be discovered.

> There are major advantages [to open source] in the sense that hopefully we will have people across the world using our code. And they will find bugs, guaranteed. They will report those bugs and we can fix them and that is all going to improve the validity of the code (IW).

These technologies are justified by their proponents according to the role they play in combating brittleness. They keep Fluidity reliable and stable for its users. The code is changing all of the time, but if you got a certain result last week, if it doesn't work this week you know exactly the version of the code in which it did work and you know who is responsible for the change, and the automated testing system probably already alerted you to the problem as soon as it happened, dampening the risk of future delays to your investigations.

On the other hand, the introduction of all these systems has transformed the working style of the group. Together they regulate everyday research. For every new piece of code, a test must be written, the user guide must be updated, and a review must take place. Failed tests must be responded to. Bugs must be logged, cross-checked, assigned, fixed, and tested. Example simulations for training purposes must be generated and documented. Requests for help on the email lists and chat channel must be responded to. Because the style and structure of the code must be kept consistent, the group started having weekly developers' meetings in order that everyone can be kept up to date with all the other research projects that are currently going on and so that proposed changes and additions can be discussed before they are implemented. These meetings must be attended.

This tendency towards standardization and conformity fits with classic treatments of bureaucracy in the social sciences (for example Merton 1957, pp.195–206). Work with old Fluidity, a smaller code with fewer collaborators, could be quite individualistic, in the sense that scientists were free to develop their own working practices, styles of research and ways of communicating. They could keep each other informed of what they were doing through informal means. In contrast, the social environment of the new Fluidity is much more rule-bound, with extra time investments required in order to keep each supporting system up to date and running smoothly. It can be understood as bureaucratic because the group accumulates strong norms pertaining to "correct" ways of doing things: a comprehensive set of procedures that define how research should be carried out exist and must be adhered to.

Not surprisingly, I would regularly observe friction between scientists with somewhat polarized attitudes to the transition. Some felt that these additional systems and processes were too extensive, that the workability they facilitated was bought at the high price of leaving too little time for doing the work that they saw as important, the research itself.

> There are some people here who are very interested in how the code is written so they like to have all these meetings and discuss things all the time and update people. I understand you need various things in place. But from a personal perspective I just want to do science (IM)

The result is a broad division within the research group, between on the one hand, those who self-identify as programmers and who are driven by epistemic interest in the code itself, and on the other hand, those who regard programming as a means to an end (this kind of division of orientation is common [see Merz 1999; Sundberg 2008]). On matters of programming, the former tend to exert authority, even in cases where they are

more junior in the official organizational structure. This created tension from time to time when the programmers felt that the systems were being neglected, when others failed to turn up to development meetings, or when it was discovered that sections of the user guide were not being kept up to date. On the other side, the non-programmers occasionally addressed the division through light-hearted jibes. For example, IW would refer to his colleagues as a kind of police force, because he would often get in trouble for "breaking the laws."

> There are people … I call them "the code police"… These are people that really understand the code to a level that others don't. Any changes that aren't suitable they will say "look, that is useless," often with my changes, and say "do it this way, do it that way" (IW).

While the expression of these small frustrations was a regular feature of my interaction with these scientists, they all recognized that these systems were essential for the maintenance of a code of Fluidity's nature. "Sometimes there is a feeling of being too software engineering focused but in the long run you need to do that, otherwise you will end up with codes that are impossible to run" (CA). While interviewees would make their resistance known to me, none of them went so far as to claim that Fluidity could be better run in another manner.

8. Programming Systems Products

The tension between advocates of regulated practices and those who voice resistance is a reflection of a general pressure that software exerts. While the agile practices discussed above are relatively recent, the amplification of effort involved in managing large software projects has been recognized in the software engineering literature for many years. In his classic essay "The Tar Pit," Frederick Brooks analyzes precisely the kinds of frustration that those like IM express when faced with the routines and processes of large code development (1995). He noted that programmers working on big projects commonly feel that if only they could be left alone, if only they could just be free of all this bureaucracy, this feeling of wading through a tar pit of daily routine, of meetings, processes and management, then they could be so much more productive. The fresh air of freedom and they could write many more lines of code; they could create something amazing. If brittleness is one "material" property of software environments, the tar pit is another, the feeling of density and sluggishness imposed by the additional organizational routines that grow up around the central task of writing big software. "One occasionally reads newspaper accounts of how two programmers in a remodeled garage have built an important program that surpasses

that best efforts of large teams. And every programmer is prepared to believe such tales, for he knows that he could build *any* program much faster than the 1000 statements/year reported for industrial teams" (1995, p.4). So, Brooks asks, "[w]hy then have not all industrial programming teams been replaced by dedicated garage duos?" (1995, p.4)

The problem, says Brooks, is that this way of thinking is mistaken about what it is that is actually being created in these different contexts. What gets written in the garage is a program. A program "is complete in itself, ready to be run by the author on the system on which it was developed" (1995, p.4). It does the core task, and may do it exceptionally well. But it works for that person, on that computer. What the big organizations are after is something quite different: a "programming systems product." This is not merely a big code. It is a code that "can be run, tested, extended or repaired by anybody… usable in many operating environments, for many sets of data… written in a generalizable fashion … [with] thorough documentation" (1995, pp.5–6). It also must conform with standard interface design, with control on memory usage, tested in all permutations with multiple other systems with which it is going to need to be able to coexist and interact (1995, p.6). Brooks' estimate is that to create a programming systems product requires *nine times as much work* as a program, and it is this amplification of effort that gives the tangible feeling of sluggishness consonant with working on big software compared with working on an individual pet project.

Brooks' essay is very helpful for understanding what took place at AMCG during the transition from the old to the new versions of Fluidity. It is also helpful for understanding the field more broadly. There is a fundamental division in computational science between research that is done with programs, and what, on the other hand, gets done with programming systems products. A lot of computational science is done in small groups, with relatively short-term goals for the software, and a more flexible attitude to its longevity and portability. This software is written in the scientific equivalent of a garage (probably the office of a principal investigator with a couple of his or her Ph.D. students, maybe a couple of postdocs). In most of these cases there is no need to create a programming systems product. You don't need to worry about new people being able to get on board quickly because the whole team was on board from the start. You don't need to worry about other groups using the software because they could just write a similar application for themselves from scratch. You don't need the software to run on any computer but the facility that you have access to for that project, and for which you have been writing from the start. This software is a relatively disposable means to an end. Its endurance is a "nice to have" but the real output is discursive.

On the other hand, a significant subset of problems can never be tackled with small programs because the kinds of systems that are to be the focus of research are so complex that a large software framework is a necessity. Some geophysical problems involve multiple interacting processes, operating on many scales. Software above a certain size requires a programming systems product approach if it is to stay habitable, to remain material for productive research. A big team needs to be coordinated around it. The project will take long enough that some of the founders will leave and new people will have to be brought on board part way through. It therefore has to be standardized and well documented. Hardware systems will be replaced. Adjacent software systems such as compilers and operating systems will be updated. The developers must be able to respond to these changes without huge headaches. With complex geophysical problems, the time it takes to build in all the necessary functionality and to validate all the parts of the model is so great, that the software itself needs future-proofing. It is not just the money; careers are being invested. The legacy needs to be more than the paper output of publications. It extends to the software itself, itself a platform for further studies into the future. In these cases, software is not just a means to an end. It is an output of scientific activity in its own right, and, for that to be the case, the software and the frameworks that surround it take a different form to small-scale endeavors. It must be legible, extendible, and widely compatible, and this carries with it significant constraints on daily routine.

9. Conclusion

Both brittleness and the accumulation of supporting processes make demands on time, whether it is time spent figuring out how to make changes or fix bugs, or time spent adhering to new administrative routines for tracking changes and logging bugs. These demands emerged as Fluidity grew, and we can expect them to present general pressures on computational science research groups in many different fields. We can call them "material" properties because they emerge at the point where a system of writing exceeds its legibility, and becomes an artefact known through use and manipulation as much as it is known through interpretation as a text. Tendencies towards brittleness and bureaucracy could be discounted as merely practical, having little bearing on the legitimacy of discourse. But they play a major role in conditioning the investigations through which such discourse is generated in the first place. They are the geological processes shaping the landscape of possibilities that scientists navigate when they put their techniques into action.

And more than being just properties of scientific software, these pressures also serve as drivers of motivation. Research in science studies has

explored the axes of desire, emotion and excitement that drive research (Knorr-Cetina 2001, p.182). The picture this provides, however, needs to be filled out by exploring the other kinds of concern that cut across the epistemic, concerns with maintaining equipment under conditions of stress, of holding things together in arrangements that threaten to break down. These technical motivations are not isolated from epistemic motivations; they gain their force from the latter. The focus on the framework, on keeping it workable, portable, durable, arises from the intersection of the properties of expanding software with its conditions of application as a particular kind of medium for research.

References

Bachelard, G. 1984. *The New Scientific Spirit*. Boston: Beacon Press.

Bailer-Jones, D. M. 2003. "When Scientific Models Represent." *International Studies in the Philosophy of Science* 17 (1): 59–74.

Basili, V. R. et al. 2008. "Understanding the High-Performance-Computing Community: An Software Engineer's Perspective." *IEEE Software* 25 (4): 29–36.

Beck, K. et al. 2001. *Manifesto for Agile Software Development*. http://agilemanifesto.org/ [Accessed 16 December 2014].

Booch, G. 2008. "Measuring Architectural Complexity." *IEEE Software* 25 (4): 14–15.

Bourdieu, P. 1977. *Outline of a Theory of Practice*. Cambridge: Cambridge University Press.

Brooks, Jr., F. P. 1995. "The Tar Pit." Pp. 3–12 in *The Mythical Man-Month: Essays on Software Engineering*. Boston: Addison-Wesley.

Easterbrook, S. M., and Johns, T. C. 2009. "Engineering the Software for Understanding Climate Change." *Computing in Science Engineering* 11 (6): 65–74.

Evens, A. 2006. "Object-Oriented Ontology, Or Programming's Creative Fold." *Angelaki* 2 (1): 89–97.

Gabriel, R. P. 1996. *Patterns of Software: Tales from the Software Community*. Oxford: Oxford University Press.

Gibson, J. J. 1986. *The Ecological Approach to Visual Perception*. Hillsdale: Lawrence Erlbaum Associates.

Giere, R. N. 2004. "How Models Are Used to Represent Reality." *Philosophy of Science* 71 (5): 742–752.

Goody, J. 1986. *The Logic of Writing and the Organisation of Society*. Cambridge: Cambridge University Press.

Guala, F. 2002. "Models, Simulations, and Experiments." Pp. 59–74 in *Model-Based Reasoning: Science, Technology, Values*. Edited by L. Magnani. New York: Kluwer Academic.

Heymann, M. 2006. "Modeling Reality: Practice, Knowledge, and Uncertainty in Atmospheric Transport Simulation." *Historical Studies in the Physical and Biological Sciences* 37 (1): 49–85.

Justi, R., and J. Gilbert. 2003. "Models and Modelling in Chemical Education." Pp. 47–68 in *Chemical Education: Towards Research-based Practice*. Edited by J. Gilbert, O. de Jong, R. Justi, D. F. Treagust, and J. H. Driel. Dordrecht: Kluwer Academic Press.

Keller, E. F. 2003. "Models, Simulation, and 'Computer Experiments'." Pp. 198–215 in *The Philosophy of Scientific Experimentation*. Edited by H. Radder. Pittsburgh: University of Pittsburgh Press.

Kleindorfer, G. B., L. O'Neill, and R. Ganeshan. 1998. "Validation in Simulation: Various Positions in the Philosophy of Science." *Management Science* 44 (8): 1087–1099.

Klein, U. 2002. *Experiments, Models, Paper Tools: Cultures of Organic Chemistry in the Nineteenth Century*. Stanford: Stanford University Press.

Knorr-Cetina, K. 2001. "Objectual Practice." Pp. 184–197 in *The Practice Turn in Contemporary Theory*. Edited by K. Knorr-Cetina, Theodore R. Schatzki, and Eike von Savigny. London: Routledge.

Knuuttila, T. 2011. "Modelling and Representing: An Artefactual Approach to Model-Based Representation." *Studies in History and Philosophy of Science* 42: 262–271.

Knuuttila, T. 2005a. "Models as Epistemic Artefacts: Toward a Non-Representationalist Account of Scientific Representation." *Philosophical Studies of the University of Helsinki* 8.

Knuuttila, T. 2005b. "Models, Representation, and Mediation." *Philosophy of Science* 72 (5): 1260–1271.

Lahsen, M. 2005. "Seductive Simulations? Uncertainty Distribution Around Climate Models." *Social Studies of Science* 36 (6): 895–922.

Marick, B. 2008. "A Manglish Way of Working: Agile Software Development." Pp. 185–201 in *The Mangle in Practice: Science, Society, and Becoming*. Edited by A. Pickering and K. Guzik. Durham: Duke University Press.

Matsumoto, Y. 2007. "Treating Code as an Essay." Pp. 477–481 in *Beautiful Code: Leading Programmers Explain How They Think*. Edited by A. Oram and G. Wilson. Cambridge: O'Reilly.

Merton, R. K. 1957. *Social Theory and Social Structure*. Glencoe: Free Press.

Merz, M. 1999. "Multiplex and Unfolding: Computer Simulation in Particle Physics." *Science in Context* 12 (2): 293–316.

Morgan, M. S. 2012. "Models as Working Objects in Science." Paper presented at the Models and Simulations 5 Conference, June 14–16.

Morrison, M. 1999. "Models as Autonomous Agents." Pp. 38–65 in *Models as Mediators: Perspectives on natural and social science*. Edited by M. S. Morgan and M. Morrison. Cambridge: Cambridge University Press.

Netz, R. 2003. *The Shaping of Deduction in Greek Mathematics: A Study in Cognitive History*. Cambridge: Cambridge University Press.
Oreskes, N., K. Shrader-Frechette, and K. Belitz. 1994. "Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences." *Science* 263 (5247): 641–646.
Ostrom, T. M. 1988. "Computer Simulation: The Third Symbol System." *Journal of Experimental Social Psychology* 24: 381–392.
Parker, W. 2009. "Does Matter Really Matter? Computer Simulations, Experiments, and Materiality." *Synthese* 169 (3): 483–496.
Parker, W. 2011. "When Climate Models Agree: The Significance of Robust Model Predictions." *Philosophy of Science* 78 (4): 579–600.
Polanyi, M. 1983. *The Tacit Dimension*. Gloucester: Peter Smith.
Rheinberger, H.-J. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford: Stanford University Press.
Segal, J., and C. Morris. 2008. "Developing Scientific Software." *IEEE Software* 25 (4): 18–20.
Suárez, M. 2010. "Scientific Representation." *Philosophy Compass* 5 (1): 91–101.
Sundberg, M. 2008. "The Everyday World of Simulation Modeling: The Development of Parameterizations in Meteorology." *Science, Technology & Human Values* 34 (2): 162–181.
Winsberg, E. 2009. "A Tale of Two Methods." *Synthese* 169 (3): 575–592.