'DOES IT WORK?': THE UNFORESEEABLE CONSEQUENCES OF QUASI-FAILING TECHNOLOGY

Federica Frabetti

Monsters vs. Aliens

'Fighting an alien robot? That was me! And it was amazing!', boasts Susan Murphy soon after having defeated an alien robot probe in San Francisco, with the help of a gelatinous blue blob and a gay fish-ape hybrid. She gleefully proceeds to enumerate all the different ways in which being a monster is an extremely appealing and rewarding status for an American girl of her age. All the while, the group of freaks that surround her – which includes a mad scientist and a perambulating insect chrysalis – marvel at the discovery of their own virtues and talents.

I want to offer a brief reading of the computer-animated 3D feature film from DreamWork Animation and Paramount Pictures, *Monsters vs. Aliens*, as a kind of anticipation of the argument of this article. In this film (released in March 2009) Susan Murphy, a young woman from Modesto, California, is hit by a radioactive meteor on the day of her wedding, thus absorbing a rare substance called quantonium which mutates her into a giantess. Immediately captured by the US military and classified as a 'monster', she is imprisoned in a top-secret facility headed by General W.R. Monger where other 'monsters' are also kept in custody. Among them are B.O.B. (Bicarbonate Ostylezene Benzonate, an indestructible gelatinous blue blob without a brain), Dr. Cockroach, Ph.D. (a mad scientist with a giant cockroach's head), the Missing Link (a 20,000-year-old amphibious fish-ape hybrid) and Insectosaurus (a 350-foot grub). When an alien named Gallaxhar attacks the Earth with his gigantic robotic probes and an army of clones of himself, General Monger persuades the president of the United States to deploy the monsters as military weapons. Having accepted the mission with the promise of freedom if they succeed, the monsters manage to destroy the alien robotic probe that Gallaxhar has sent to San Francisco. During the fight Susan discovers that she possesses an unexpected strength and that she is also invulnerable to Gallaxhar's weapons. Having been freed, Susan happily returns to Modesto - only to be rejected by her fiancée (who claims that he cannot be married to a woman who overshadows him). In the meantime, her monstrous friends unwittingly cause panic in the neighbourhood. Initially sad and dispirited, Susan suddenly realizes that becoming a monster has actually enriched her life, and she fully embraces her new 'amazing' lifestyle and her newly formed bond with the other monsters. After a final epic fight Susan and her gang completely defeat Gallaxhar and his cloned army, and are eventually acclaimed as heroes. In the last scene of the film, they are alerted to the fact that in the surroundings of Paris a snail has fallen into a nuclear power-plant and is growing into a giant due to nuclear irradiation. They then fly off on a mission to protect the Earth from the new enemy.

What is particularly interesting about *Monsters vs Aliens* is that in this movie the monsters function first and foremost as a figure of the unexpected consequences of technology. Not only do they all *come into existence* as the unpredictable outcomes of experiments gone wrong (B.O.B. was mistakenly created by injecting a genetically-modified tomato with a chemically-altered ranch dressing; Dr. Cockroach ended up with an insect head and the ability to climb walls while subjecting himself to an experiment in order to gain the longevity of a cockroach; Insectosaurus, originally a one-inch grub, was transformed into a giant after being accidentally invested by nuclear radiation, and while the mad scientist is the figure of the experiment gone wrong *par excellence*, even the Missing Link could not have been found frozen in a lagoon and thawed out by scientists without some help from technology).[1] Even more importantly, the monsters are also 'domesticated' – or rather, they are kept under custody by the American government and later on transformed into weapons. In other words, the film seems to imply that technology needs to be controlled in order to be made useful – that is, it has to be made into a tool.

Nevertheless, in order to be successfully deployed as weapons, monsters must be released from custody – or, in order to be 'used', technology must be set free. Yet once it is set free, technology seems to escape its own instrumentality. Indeed, it is by fighting Gallaxhar that Susan discovers her unexpected physical strength, while during the final battle against the aliens Insectosaurus apparently dies, only to undergo a metamorphosis from a chrysalis into a beautiful butterfly. Ultimately, though, the monsters are still kept under control: they constitute an American military team - albeit a very special one. It is here that aliens find their place in the film narrative: a relationship which would otherwise be quite uncomplicated (humans detain and domesticate dangerous monsters) finds its third term in the aggressive threat from the outside. Aliens provide an enemy and help construct the narrative of the American fight for democracy against (alien) totalitarian regimes. Even though it occasionally makes fun of the American government (General W.R. Monger's name is a pun on the word 'warmonger', and the inept president of the United States is always on the verge of launching a nuclear attack by pressing the wrong button), the film still embraces a narrative that legitimates the Unites States as the world superpower.[2]

However, the most interesting point of the film is to be found at the end. In the final scene the monsters set off to Paris to fight a gigantic snail, which has broken into a nuclear plant – but should the snail be perceived as an alien or a monster? Since it is presented as a threat against which the monsters are supposed to fight, it must be an alien. And yet, since clearly it is an unexpected effect of technology (actually, accidental nuclear irradiation is one of the most common origin stories of superheroes and is very similar to Insectosaurus' story), the snail must be a monster and in principle it should not be fought but rather helped out or maybe even recruited as part of the team. With a revealing lapse, a Wikipedia entry (http://en.wikipedia.org/wiki/Monsters_vs_Aliens) recounts how at the end of the film 'the monsters are alerted to a *monster attack* near Paris and fly off to combat the new menace' (italics mine). In Derridean terms, it could be said that the snail is the incest taboo of *Monsters vs Alien*: the locus where the distinction between monsters and aliens becomes untenable; it is the 'point of opacity' of the film, or the point where the film narrative undoes itself.[3] But why is it important to think about the untenability of the distinction between monsters and aliens, and, ultimately, about the distinction between usable, domesticated, *functioning* technology on the one hand, and failing, unpredictable technology, or technology out of control, on the other?

What I want to argue in this article is that unusability, failure and the capacity for generating unexpected consequences are in fact constitutive of technology. Indeed, technology cannot exist without failure. It cannot be separated from its constitutive fallibility, which, importantly, also drives its growth. Yet technology also exists only inasmuch as such a separation is continually reasserted – even though it keeps becoming undone. What is more, I want to argue that the failure of technology is tightly connected with the concept of instrumentality. Commonsensically, failing technology is a technology that does not *work,* or that does not work *as expected*. In doing so, technology escapes its own conception as an instrument, a tool which we can use, control and master. In other words, it exceeds its own instrumentality and gives rise to the unpredictable. And yet, it is precisely the capacity of technology for *not* working – that is, for generating unexpected consequences – that ultimately makes it possible for technology to work.

Moreover, as I will show in a moment, every failure requires a decision in order to be constituted *as* a *failure* – that is, a decision about what behaviours are considered 'expected' or 'acceptable' for a given instantiation of technology. From a technical point of view, 'failure' – that is, a malfunction – is something that needs to be fixed. However, there exists a widespread recognition in media and cultural studies that a rethinking of contemporary technology is needed today. For those of us who want to think about technology differently by asking different questions of technology from the technical ones, perfectly functioning technologies might not be the most interesting ones to look at. Indeed, I want to argue here that technology is at its most revealing precisely when it does *not* work – or, even better, when it is unclear, to common users and even to technical experts, *whether* it is working or not. I call these moments, which occur more frequently than one might think at first glance, 'quasi-failures', and these instances of technology - 'quasi-failing technologies'. I will discuss some examples of failing and quasi-failing technologies in the further parts of this article, in order to show to what extent and in what way they can contribute to a different cultural and political understanding of technology. What is at stake in this discussion is ultimately the possibility of a non-functionalist engagement with technology. I am thus posing two questions: Can we say something about technology that is not an explanation of how it does or does not work? Can we give a non-functionalist answer to the question 'what does technology do?'?

These questions are extremely important for the investigation of what are commonly named 'new' or 'digital' technologies in the field of media and cultural studies. When approaching new technologies, media and cultural studies has predominantly focused on the intertwined processes of production, reception, and consumption - that is, on the discourses and practices of new technologies' producers and users. From this perspective, even a technological object as 'mysterious' as software is addressed by asking how it has been made into a significant cultural object. For instance, in his 2003 article on software, Adrian Mackenzie demonstrates the relevance of software as a topic of study essentially by examining the new social and cultural formations that surround it (Mackenzie, 2003). An analogous claim is made by Lev Manovich in his recent book, *Software Takes Command* (2008), where, while arguing that media studies has not yet investigated 'software itself', and advancing a proposition for a new field of study that he names 'software studies', Manovich is actually adamant that software studies should focus on software as a cultural object - or, in Manovich's own terms, as 'another dimension in the space of culture' (Manovich, 2008: 4). Software becomes 'culturally visible' only when it becomes visual – namely, 'a medium' and therefore 'the new engine of culture' (4). On the other hand, when addressing the workings of technology cultural investigations of new technologies tend to draw on the explanation of how a specific instance of technology functions as it can also be found in technical literature (see, for instance, Galloway, 2004). Although I recognize that the above perspectives remain very important and politically meaningful for the cultural study of technology, I suggest that they should be supplemented by an alternative, or I would even hesitantly say more 'direct', investigation of technology. I want to argue that it is possible to engage in a closer, even intimate relationship with technology by focusing on the capacity of technology to generate the unexpected. Such cautious intimacy would also be crucial in developing a political understanding of technology. It would still aim at demystifying technology and at dispelling what Bernard Stiegler has called its 'opacity' (Stiegler, 1998) in order to make contemporary technology thinkable, thus ultimately enabling us to make decisions about technologies that increasingly escape our understanding. And yet, such an approach would remain aware of the fact that our access to technology is always mediated: it would definitely not assume that technology can be made totally transparent. It would also keep questioning the very nature of our 'intimate' engagement with technology itself.

In order to do further such an alternative understanding of technology, I want to start here by looking at what is possibly the least accessible (yet most pervasive) of digital technologies – the one we commonly refer to as 'software'. Firstly, I want to explore in what way software's fallibility was perceived by the emerging discipline of Software Engineering at the end of the 1960s -precisely when the term 'software' was becoming popular both within and without the technical realm. Secondly, I want to examine how, starting from the second half of the 1990s, such fallibility has been 'put to work' in the open source movement.

Calculating the Unforeseeable in the Cold War Years

The discipline of Software Engineering emerged as a strategy for the industrialization of the production of software at the end of the 1960s. The first two conferences on Software Engineering were convened by the NATO Science Committee in 1968 and 1969, in Garmisch (Germany) and Rome (Italy) respectively. They involved all the so-called 'founding fathers' of Software Engineering (the well-known computer scientists of the time, such as Edsger W. Dijkstra and Peter Naur), dealt with many of the topics that still constitute the agenda of Software Engineering today and were accurately documented through the publication of detailed proceedings. These two conferences – especially the one held in Garmisch - are considered the founding moment of Software Engineering as both an academic discipline and a methodology for software production. Actually, the Garmisch conference report can be thought of as the foundational narrative for the field. What I want to propose in what follows is that, in the late 1960s, Software Engineering established itself as a discipline precisely through an attempt to control the constitutive fallibility of software-based technology.[4]

Historically, Software Engineering emerged from a crisis, the so-called 'software crisis' of the late 1960s. As Brian Randell – editor of the reports of the 1968 and 1969 conferences – recalled later on in his article 'Software Engineering in 1968', one of the most significant aspects of the NATO conferences was the willingness of the participants to admit 'the extent and seriousness' of the software problems of the time (Randell, 1979: 1). For instance, during the Garmisch conference Dijkstra reportedly stated that '[t]he general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement' (Naur & Randell, 1969: 121). Terms such as 'software crisis' and 'software failure' were largely used at the NATO conferences on Software Engineering, and for this reason many of the participants viewed that conference as a turning point in their way of approaching software and in their work in the field. Indeed, with the NATO conferences, software began to be conceptualized *as a problem* – and the 'software crisis' was constituted as a point of origin for the discipline of Software Engineering. From the very beginning, the participants in the Garmisch conference acknowledged that they were dealing with 'a problem crucial to the use of computers, viz. the so-called software, or programs, developed to control their action' (Naur & Randell, 1969: 3). The very first lines of the Garmisch report establish a clear relationship between software and control, while at the same time characterizing this relationship, as well as software itself, as problematic. But why was software 'problematic' in the late 1960?

The problems that the Garmisch conference attempted to address were mainly related to 'large' or 'very large' software-systems – that is, systems of a certain complexity whose development required a conspicuous effort in terms of time, money and the number of programmers involved.[5] As Randell comments in his recollections about the Garmisch conference (thus explaining NATO's interest in Software Engineering), 'it was the US military-industrial complex that first started to try and develop very large software systems involving man-millennia of effort' (Randell, 1979: 5). Randell also mentions a paper presented by Joseph C. R. Licklider as a contribution to the public debate around the Anti-Ballistic Missile (ABM) System (a complex project which contemplated the development of enormously sophisticated software) and eloquently titled 'Understimates and Overexpectations'. In his paper Licklider provides a vivid picture of the gap between the military's goals and their achievements. He declares: '[a]t one time, at least two or three dozens complex electronic systems for command, control and/or intelligence operations were being planned or developed by the military. Most were never completed. None was completed on time or within the budget' (Licklider, 1969: 118).

Even more importantly, Randell adds the following comment:

> I still remember the ABM debate vividly, and my horror and incredulity that some computer people really believed that one could depend on massively complex hardware and software systems to detonate one or more H-bombs at exactly the right time and place over New York City to destroy just the incoming missiles, rather than the city or its inhabitants. (Randell, 1979: 5).

Here Randell's 'horror' at the excessive self-confidence of some software professionals stems from the connotative association between technology, catastrophe and death in a cold-war scenario. As we shall see in a moment, horror – a powerful emotion - is the result of the anticipation of the consequences of technology combined with the awareness of its intrinsic fallibility.

However, by the late 1960s large-scale systems were not unique to the military scene. Computer manufacturers had started to develop complex operating systems. Specialized real-time systems were also being developed, such as the first large-scale airline reservation system, the American Airlines (SABRE) system. The costs incurred in developing these systems were immense and they were very much in the public's eye. Moreover, some of these systems (such as TSS/360, and even IBM OS/360) kept performing poorly notwithstanding the vast amount of resources lavished on them by their manufacturers – and the professionals involved in these projects felt the pressure of the public opinion. The Garmisch conference report was produced expressly to serve as an instrument for managers of the private and public sectors and policy makers to anticipate and evaluate the consequences of technology in time. The participants in the Garmisch conference viewed society at large as mainly concerned with the problem of the reliability of software and with its costs, and they measured the relation between software and society in terms of 'impact'. And yet – and this is an extremely important point for the investigation of software failure - it is precisely this opposition between society and technology that seems not to hold everywhere in the Garmisch report. For instance, participant E. E. David describes the process of software growth according to the report in the following terms:

> In computing, the research, development, and production phases are often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. … This situation is familiar in all fields lacking a firm theoretical base. Thus, there are good reasons why software

> tasks that include novel concepts involve not only uncalculated but uncalculable risks. (Naur & Randell, 1969: 15 f.)

David focuses here on the pace of software growth. The competition between computer manufacturers forces software professionals to confuse ('telescope') research and production, which should remain separate. Therefore, the uncertainties which are typical of research (here intended as the development of innovative software) spread to production. David's metaphor opposes 'leaps' to 'steps'. The leap is for him a dangerous way to move forward, motivated by the lack of knowledge. The step-by-step approach would be a safer way - not to slow down the growth of software, but to make the speed of such growth more manageable. One must be reminded once again here that the participants in the Garmisch conference had to face some major doubts concerning large-scale software systems: were such systems actually feasible? In David's terms, the question could have been reformulated as follows: was the speed of software growth actually manageable? Importantly, David's statement attributes the need for taking big leaps forward to the lack of a 'firm theoretical basis': in other words, the inability to estimate the feasibility of a software project in a reliable way leads to the impossibility of carrying it out step by step, and ultimately to its failure. The failure of a software project then seems to be related to the failure of the management of time.

According to David, software professionals are fundamentally concerned not just with risk (that is, the possibility of failure) but also with 'uncalculated' and 'uncalculable' risks. It seems quite understandable that certain risks cannot be calculated due to the lack of accurate knowledge. What is really surprising though is David's use of the expression 'uncalculable'. It is not quite common for software professionals, and for engineers in general, to acknowledge that a technical project involves uncalculable risks. Although the participants in the Garmisch conference must not have been aware of this fact, the concept of the calculability of time has a distinct Heideggerian echo.[6] I will come back to this point in a moment. However, for David the concepts of risk and calculability are both related to the future: estimates are the expression of a calculability of the future, they actually *presuppose* the calculability of the future. And it is precisely this faith in the calculability of time, and therefore in the feasibility of software projects, that is put into question in the Garmisch report (as well as in the narrative of the 'software crisis' as the source of technological 'horror').

At this point I want to posit the following question: to what extent can the incalculability which is lamented by David be linked to the 'unforeseen consequences' that for Jacques Derrida are always implicit in contemporary technology? One must be reminded here how in a dialogue with Bernard Stiegler published in *Ecographies of Television,* Derrida argues that the acceleration of technological innovation in the contemporary world constitutes a 'practical deconstruction' of the instrumental conception of technology (Derrida & Stiegler, 2002: 45). It is true that in the contemporary world technological innovation is massively appropriated by multinational corporations and nation states, by means of their 'research and development' and 'defence' departments, and that technological innovations are constantly programmed to support economy. But it is also true that technological innovation still gives rise to unforeseen effects. Derrida even propounds that the greater the attempt to control innovations, the more unforeseeable the future becomes. Such unforeseen effects ultimately deconstruct the understanding of technology as merely instrumental, as well as the perception of the human as separate from his tools and a master of them. But in what way did the participants in the NATO conferences explain the 'uncalculability' of technology?

The Garmisch conference report is dominated by a widespread recognition that the ninety-nine per cent of software systems worked – as Jeffrey R. Buxton states - 'tolerably satisfactorily' (15). Only certain areas were viewed with concern. Kenneth W. Kolence comments:

> The basic problem is that certain classes of systems are placing demands on us [software professionals] which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis – sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy. (Naur & Randell, 1969: 16)

We already know that the risky 'classes' of systems are large-scale and real-time ones. Nevertheless, this passage seems to take the argument a step further and relate the uncalculability of software development to certain demands posed by society that go beyond the technological capabilities of the time.

In other words, not only did the conference participants feel the pressure of social demands on them; they also felt that software development reached its point of crisis when society pushed the boundaries of state-of-the-art technology. But did these demands come from society or from technology itself? Here I want to make the suggestion that such a question is at work in the whole of the Garmisch report and that it silently destabilizes the separation between the technical and the social. Actually, it is precisely when dealing with the issue of the responsibility for the technological risk that the conference participants seem to be confronted with the impossibility of separating technology from society. For instance, Ascher Opler states:

> I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity? Is it the manufacturer's fault for producing them or the users' for demanding them? One shouldn't ask for large systems and then complain about their largeness. (Naur & Randell, 1969: 17)

Opler's passage is intriguingly ambiguous. He asks whether the responsibility for the rate of the growth of technology must be attributed to the users or producers of technology. The undecidability of this dilemma leaves its mark on the field of Software Engineering and especially on its relationship with technological failure. On the one hand, the participants in the Garmisch conference seem to acknowledge that risks are implicit in software, and that software fallibility is unavoidable. This is what David and Fraser state: '[p]articularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death' (Naur & Randell, 1969: 16). On the other hand, it is claimed that risks can be avoided if an appropriate and effective 'theory' of the development of software was to be produced. From this latter point of view, the approach to software development must be 'systematic' (Shaw, 1989), and therefore it must become a form of engineering. However, these two points of view are entangled and one does not exist without the other.

As a result, Stanley Gill concludes: '[i]t is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of technology unless the very considerable risks involved can be tolerated' (Naur & Randell, 1969: 18). This quotation might sound like an attempt to discharge the responsibility for technological risk on society. In fact, it requires deeper analysis, since in what way could policy makers evaluate risks that they do not know? Software professionals are the ones who are expected to have such knowledge. A Habermasian answer might suggest that policy makers should be better informed of technological risks and able to discuss them freely (Habermas, 1991). But what Gill is actually saying here is that society shall not make demands that can be met only by exceeding the current state of technology. Here we are confronted with one of the 'points of opacity' – as Derrida (1980) would have it – of the foundational narrative of Software Engineering. Indeed, it seems to me that the irreconcilability of these two aspects – and therefore the necessity of calculating incalculable risks, and of attributing responsibility for them – is a point where Software Engineering 'undoes itself' precisely at the moment of its constitution. What Gill means here is that society needs to take responsibility for an incalculable risk. The real problem here is the incalculability of the speed of technological growth - that is, of the rate at which the state of technology is exceeded.

In sum, at the end of the 1960s Software Engineering as a discipline with a theoretical foundation is called for in order to avoid the (unavoidable) fallibility of technology – a fallibility that constitutes the risk posed by technology, or, better, technology *as* a risk. This point of opacity suggests that Software Engineering establishes itself *as a theory* of technology by expelling fallibility from technology – but such a fallibility (the unexpected consequences of technology) is intrinsic to technology itself, and is exactly what allows Software Engineering to exist (that is, the reason why Software Engineering is called for). In other words, Software Engineering performs an impossible expulsion of constitutive failure from technology, while simultaneously establishing itself as a discipline with this move. Since such an expulsion is performed through the calculation of time, it can also be said that in Software Engineering the calculability of time is undone in its very constitution. Going beyond Stiegler's concept of the dis-adjustment between technology and society (Stiegler, 2003), I also want to suggest that society is instituted in the Garmisch report as that which places risky demands on technology – while at the same time the report declares technology as constitutively fallible, as something that intrinsically incorporates unforeseen consequences. Therefore, the projection of the fallibility on society - that is, on the demands that society poses to technology - is the way in which the conference participants both assume and discharge responsibility for the technological risk: they cannot actually maintain the boundary between technology and society, because this boundary keeps becoming undone. This is why I said earlier that (in Heideggerian terms) Randell's 'horror' is the result of anticipation *plus* the fallibility of technology. In a way, it can be said that, contrary to Heidegger's understanding of the relationship with death as constitutive of a temporality which is more 'authentic' than the temporality of calculation, in Software Engineering the question of death (for instance, the death of New York's inhabitants caused by a ballistic device gone wrong) is dealt with *as* a problem of calculation.

Are You Experienced? Clumsy Users and Dumb-Proof Technologies

In the Garmisch conference report 'the user' makes its appearance as a problematic figure towards whom software developers have ambivalent feelings. On the one hand, J. N. P. Hume suggests that designers must not 'over-react' to individual users – that is, in order to develop an effective and usable software system, they must identify the requirements 'common to a majority of users' and focus on them (Naur & Randell, 1969: 40). On the other hand, J. D. Babcock argues for the intelligence of the users. He comments: '[t]he users are the people who do our design, once we get started' (40). In doing so, Babcock awards 'the users' an essential role in the process of software development various decades before the emergence of cooperative Human-Computer Interface (HCI).[7] However, the conference participants express a general discomfort about interacting with 'the user'. Manfred Paul describes the user as someone who 'does not know what he needs', but he couples this with another kind of ignorance: users are actually 'cut off from knowing what is or what might be available' (40). And Al Perlis adds: 'Almost all users require much less from an operating system than is provided' (40). In these two passages users are understood alternately as unable to understand their own needs – and thus unable to pose clear requests to technology, and as overwhelmed by the technological offer – and thus incapable of making the most of the functionalities provided by technology.

These complaints about 'users' are a familiar feature not just of Software Engineering but also of the general approach of software developers to their non-technical counterparts (see, for instance, Bolter, 1984). However, it would be reductive to interpret such complaints merely in terms of the difficulties encountered by software practitioners in communicating with non-technical users. Importantly, J. W. Smith notices that designers usually refer to users as 'they', 'them' (Naur & Randell, 1969: 40) - a strange breed living 'there in the outer world, knowing nothing, to whom nothing is owed'. He also adds disapprovingly that most designers 'are designing… for their own benefit – they are literally playing games' (40). They have no conception of validating their design, or at least of evaluating it in the light of potential use (40).

This representation of the user as someone 'out there' – someone whose 'needs' should be taken into account in order to validate software instrumentally – is particularly relevant if we are to understand how the figure of the user operates in Software Engineering. In fact I want to suggest that the 'user' and their 'needs' are part of a narrative that institutes a fictional 'origin' of the software system. As I have shown earlier on, in the Garmisch conference report 'society' is the locus of a projection of the 'demands' that are supposedly made of technology. Similarly, when conceiving a software system, software engineers understand it as the solution to some pre-existing 'problem', which is projected in the world 'out there' in order to justify the existence of software. Here I want to emphasize that the figure of the user plays an analogous role – that is, the user's needs are part of a narrative that software developers construct in order to justify the system they are developing. This is not to say that users do not really exist or that they do not express their demands in terms of what functionalities should be provided by a software system. In fact, the Garmisch conference report takes communication with users very seriously at all levels. And yet, what I want to point out is that the figure of the 'user' is positioned by the report outside the process of software development in a constant and incomplete movement of 'expulsion' of certain characteristics of software *as* 'user needs'. In Goos's words, software developers need to 'filter the recommendations coming from the outside' (Naur & Randell, 1969: 41). A double strategy is at work here, which acknowledges the importance of users while focusing on how to keep them at bay. Randell even laments the amount of time wasted on 'fending off the users' (41). Thus, 'the user' is both constituted and neutralized: while it is acknowledged that software development is set in motion by the very existence of (potential) users and that it needs their feedback, the very development of the software system acts as a form of containment of the (supposed) user's exigencies.

Even more importantly, the figure of the user is associated with the so-called 'extensibility' of software. According to Letellier, a software system should be 'extendable', or 'open-ended', thus allowing its developers to modify it in the future (Naur & Randell, 1969: 38). Moreover, as H.R. Gillette points out, 'documentation' (commonly referred to as 'user manuals') must be provided to users, whose goal is 'to train, understand, and provide maintenance' (39). User manuals are what enables users to enter an active relationship with software. Ultimately, they allow users to engage with a system whose open-endedness is inscribed in code. Therefore, documentation also constitutes a point where the capacity to take advantage of such open-endedness and to take the system into an unexpected direction is ultimately handed over to the users. This does not mean that any user can actively reprogram any system. In fact, according to the Garmisch conference report, one of the aims of software developers is to make the system 'dumb-proof' – that is, robust and resilient enough to resist 'improper' uses on the part of inexperienced and non-technical users (Naur & Randell, 1969: 40). And yet, it seems to me that the figure of the user is the locus where the instrumentality of software is both reasserted by implicitly defining it as a tool to be 'used' and opened up to unexpected consequences. The 'user' is actually a name given to a part of the process of software design. It is a field of forces that both constitutes the process of software development and destabilizes it through practices which are potentially characterized by ignorance, impropriety and the threat of failure. In the figure of the user the instability of the instrumental understanding of software and software's capacity for escaping instrumentality through the unexpected consequences it generates become apparent. Even more importantly, the ambivalent figure of the user will be at the core of many unexpected developments of Software Engineering in the 1980s and 1990s.

A Malfunction Is a Decision

'Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected', writes Eric Steven Raymond in his article of 1997 titled 'The Cathedral and the Bazaar'. This article, republished on-line many times, constitutes the Bible of Software Engineering for the open source movement. It was conceived by Raymond as an answer to Frederick Brooks' classical manual of Software Engineering, *The Mythical Man-Month* (1995), which was published in 1975 and which still remained influential in the 1980s. The title of Raymond's article is actually a pun on Brooks' famous metaphor of software development as a 'cathedral'. In 1975 Brookes was still very much preoccupied with time management, especially in relation with the organization of large groups of programmers. '[M]ost programming systems', he muses in his book, 'reflect conceptual disunity far worse than that of cathedrals', albeit they did not take centuries to build (Brookes, 1995: 42) – and yet, such disunity does not arise from 'a serial succession of master designers, but from the separation of design into many tasks done by many men' (42). Brooks' main point here is that 'conceptual integrity is *the* most important consideration in system design' (42). A software system needs to 'reflect one set of design ideas' (42), and for this reason software design needs to be structured hierarchically, so that a small group of designers is in charge of all the conceptual decisions which a larger group of programmers will then implement into code. Although a detailed discussion of Brooks' argument would be outside the scope of this article, it is worth noting that in his theory of Software Engineering Brooks attempts to control at least part of the unpredictability of software through the hierarchical organization of its development.

According to Raymond, Linus Torvald developed Linux according to a very different methodology.[8] Raymond contrasts the two models of the 'cathedral' and the 'bazaar' – where the 'cathedral' model is common to most of the commercial world, while the 'bazaar' model belongs to the Linux (and the open source) world. What Raymond calls the 'cathedral' model is in fact Software Engineering as conceived by Brooks – that is, quite a consolidated discipline with its own established corpus of technical literature. Raymond argues that the two models of the cathedral and the bazaar are based upon contrary assumptions about the nature of software development, and particularly of software debugging.

Software debugging is a late stage of software development, and is part of what in Software Engineering is generally called 'test phase' (Sommerville, 1995). Before being released to commercial users, a software system needs to be tested – namely, it is necessary to verify that the system meets its specifications, or (once again) that it works *as expected*. One of the activities involved in testing is debugging: when a test reveals an anomalous, or unexpected, behaviour of software, code must be inspected in order to find out the origin of the anomaly – namely, the particular piece of code that performs in that unexpected way. Code must then be corrected in order to eliminate the anomaly. The testing process takes time because all the functions of the system need to be tested. Furthermore, sometimes the correction of an error introduces further errors or inconsistencies into the system and generates more unexpected behaviour. Although in the phase of testing unexpected behaviour is generally viewed as an error, it is worth noting that decisions must still be made at this level. The testing team is responsible for deciding whether the unexpected behaviour of the system must be considered an error or just something that was not anticipated by the specifications (since, as we have seen earlier on, specifications are never complete) but that does not really contradict them. Errors need to be fixed by correcting code, but non-dangerous, and even useful, anomalies can just be allowed for and included in the specifications. Thus, the activity of deciding *whether an anomaly is an error* introduces changes into the conception of the system, in a sustained process of iteration.

The complexity of the above process explains why software errors are also called 'bugs'. Although the etymology of the term is uncertain, it hints at the fact that errors are often very hard to find – like the moth that Grace Hopper is said to have found trapped in a relay of the electromechanical computer Mark II in 1945, which caused many malfunctions. Locating a bug is hardly a straightforward and unequivocal process. Not only is it necessary to find out what part of code causes a malfunction, and to read it in order to find out what mistake has been made in writing it. More importantly, very often no obvious mistakes (such as misspellings) can be found because the malfunction is the result of the interaction of that piece of code with other pieces of code. Thus, more code has to be inspected, and the process tends to grow exponentially. For this reason Raymond introduces his famous aphorism that 'given enough eyeballs, all bugs are shallow' (Raymond, 2000: non-pag.). This principle is the foundation of the whole conception of open source Software Engineering, since the realization of an open source project is a collective task. Simply put, according to Raymond, Torvald maximized the number of 'person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable' (non-pag.). Raymond's passage shows how in open source the maximization of productivity is still the aim – but now programmers are prepared to risk the instability of the system, or rather, they have accepted that instability is the fastest way forward. In a way, it can be said that open source programmers feel comfortable with the idea of working on a Stieglerian device that goes faster than its own time (Stiegler, 1998) - they even use such speed to manage the project itself.[9] Moreover, they are comfortable with software anomalies, malfunctions and failures. Torvald releases different versions of the system very rapidly, because, as Raymond explains, 'given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone', or, as we have seen above, and according to what he calls 'the Linus' Law': 'given enough eyeballs, all bugs are shallow' (Raymond, 2000: non-pag.).

For Raymond it is quite obvious that 'more users find more bugs', because they all have different ways of stressing the functions of the program (for instance, inventing new uses for it). This effect is amplified when users are co-developers. It could be said that, in open source, making demands that exceed the boundaries of technology has stopped being a problem. In fact, making unexpected demands towards software seems to be the only way for software itself to grow. And this is not just because users are now empowered with the capacity for developing the system – something that to some extent they could also do in traditional Software Engineering, although there were 'gate-keepers', whose role was to 'fend off' users' requests. More importantly, in open source the development of the software system is explicitly distributed among many individuals, who produce many overlapping versions of the system. If the parallel process of the development of the system is fast enough, then the system coordinates itself. I want to suggest that this realization itself can be viewed as one unforeseeable consequence of the Software Engineering of the 1970s and 1980s.

Of course, even open source software systems need to become stable at certain points in time: any time one wants to stop being a developer and starts being a user, one must be able to 'use' the system as a tool. The stabilization of the system coincides with its instrumentalization, or, *vice versa*, instrumentality emerges with stability in time. And yet, this stabilization is not scheduled; it is not understood as the end of a certain stage of development which needs to be planned in advance and for which a deadline is established. In a way, there are no timetables, no deadlines. Stability is something that *happens* to the system, rather than being scheduled and worked toward. However, as Raymond notices in the above passage, a certain amount of control needs to be maintained over releases. Linux versions are numbered in order for potential users to choose which version to run. They can either run a more stable version (which nevertheless might present some anomalies that have not yet been solved) or 'ride the cutting edge' and run a newer version (which is likely to have been debugged further and perhaps also enriched by new functionalities, but which, for this very reason, can give rise to some more unexpected consequences). Raymond's passage attributes to users the capacity for evaluating the risks which are implicit in technology and for minimizing such risks by choosing the more stabilized version of a system. Nevertheless, he has already recognized that software always entails unforeseen consequences, to the extent that a system which is considered stable might actually lead to great surprises. What I want to suggest here is that Raymond's distinction between risky and stable systems shows that decisions regarding technology can and must be made by taking into account - rather than denying - technology's incalculability.

The Monstrous Future of Technology

In the documentary *The Net* (2003), director Lutz Dammbeck shows how obscuring the incalculability of technology leads to setting up an opposition between risk and control, and between 'good' and 'bad' technology, and ultimately to the authoritarian resolution of every dilemma regarding technology. Questions such as, 'Should technology be 'democratized'?', 'Should it be made available to everyone even when it is "dangerous"?', 'Who decides what is dangerous for whom?' are then addressed by embracing either a policy of control or a deterministic, almost paranoid fear of technology, which is also possibly combined with a Luddite stance. The film explores the complex story of Ted Kaczynski, the infamous Unabomber. A former mathematician at Harvard, Kaczynski retreated to a cabin in the wilderness of Montana in 1971. In 1996 he was arrested by the FBI under the suspicion of being responsible for the attacks carried out between 1978 and 1995 by an unknown individual nicknamed the Unabomber against major airlines executives and scientists at elite universities. The film complicates the narrative regarding the Unabomber (who was also the author of an anti-technology *Manifesto*, and an ultimate figure of resistance for those who oppose contemporary technology as a form of control) by situating him within the complex and contradictory web of the late twentieth-century technologies.

Particularly revealing is an interview with John Taylor – an ex-NASA engineer and an admirer of Norbert Wiener, the founding father of cybernetics – which shows how the idea of calculability, and the attempt to expel the unexpected from technology, was crucial for early cybernetics. Taylor recounts how ARPA (the Advanced Research Projects Agency) was set up in 1958 by the American president Eisenhower with the goal of seeking out 'promising' research projects – in Taylor's words, projects that had 'a longer term expectation associated with them'. ARPA was instituted after the launch of the Russian space probe Sputnik in 1957, which Taylor characterizes as 'a great surprise' for the United States. The American Department of Defence set up ARPA 'in the hope that we would not get surprised again like the Russian surprised us'. The ambivalence of the term 'surprise' as both risk and promise is obvious in Taylor's words: the best research projects are the ones which hold the 'promise' of 'good surprises', which will in turn prevent the enemy from surprising us in a 'bad' way. ARPA was therefore meant to 'domesticate' the potential of technology to surprise us, that is its capacity for generating the unexpected, by subjecting 'promising' projects to control. Taylor ostensibly embraces such a philosophy of control. When, during the interview, Dammbeck mentions the Unabomber, a horrified look crosses Taylor's face and, as many of his colleagues interviewed in the film do, he refuses to speak about Kaczynski, dismissing him as a terrorist and even comparing the Unabomber's *Manifesto* to Hitler's *Mein Kampf*. When Dammbeck suggests that some people such as the Unabomber might be scared by technology and asks Taylor what he is scared of, Taylor answers 'I am scared of Al-Qaeda… I am scared of cancer. But if we could find a cure for cancer, we wouldn't be afraid'. According to Taylor, fear is a matter of ignorance, of 'not knowing'. By possessing more knowledge, he pronounces via having recourse to a rather curious phrase - we could 'prohibit cancer'. Taylor's revealing formulation is the ultimate expression of a desire for the technological control over nature and for the complete calculability of the future.

The idea of cybernetics as the science of control takes up a new meaning here – one related to prediction, calculation, foreseeability.

This is particularly intriguing if one considers, as Dammbeck does, that one of the participants in the Macy Conferences (which instituted cybernetics as a discipline between 1946 and 1953), the psychologist Kurt Lewin, conceived of a project for programming humans to give them an 'anti-authoritarian personality' in order to prevent the possibility of fascism forever. Oblivious to the fact that this would be the ultimate authoritarian gesture, Lewin suggested that cybernetics could control and remap people's subconscious in order to immunize them against totalitarianism and to make authoritarian systems impossible. For him, anti-authoritarianism was first and foremost a matter of calculation, as the *control of the political future* of humanity. Ironically, drawing on Lewin's project, Henry A. Murray, one of the fathers of today's assessment centres, devised a series of tests which were supposed to highlight concealed psychological tendencies by penetrating consciousness with non-surgical means - basically LSD and other drugs. Such tests were carried out by the CIA in the late 1960s at Harvard on a group of talented young male students, among whom was Ted Kaczynski. Whether those experiments led Kaczynski to the fear of occult forms of mind control, and ultimately resulted in his paranoid terror of technology is a possibility that the film leaves open. Importantly, however, Dammbeck's film makes a suggestion that control and incalculability, risk and opportunity, are constitutive of technology. As Dammbeck himself states, the key to Kaczynski's tragedy is the fact that he is 'part of a system from which there is no escape'. He does not understand that, even isolated in a forest cabin, one is still part of the technological system (a cabin *is* a form of technology, after all), and that there is no 'outside' of technology.

Once again I want to emphasize here that in order to make responsible decisions about technology, one must be aware that technology, as well as the conceptual system on which it is based, can only be problematized from within. This is precisely what the search for the points of opacity of technology allows us to do – stepping out of a conceptual system by continuing to use its concepts while at the same time demonstrating their limitations (Derrida, 1980). This process of the problematization of technology is creative, productive and politically meaningful. In fact, it shows that, since not everything in technology can be thought or fully conceptualized within one consistent framework, and since points of opacity always remain, technology also always brings about unexpected consequences.

Perhaps the most important point of opacity that emerges from such a problematizing reading is the conceptualization of technology in terms of instrumentality. As we have seen, a sustained attempt to define software as instrumental can be found in Software Engineering. Such a definition presupposes that software is controllable, that its development and uses can be planned and that the risks and consequences implicit in software can be foreseen. Broadly speaking, this concept of software is based on the Aristotelian idea that technology is a tool that must be mastered by humans to pursue certain ends – a concept that constitutes the foundation of the general understanding of technology in the Western philosophical tradition. Consistently with this Aristotelian line of thought, not only is software defined as a tool in Software Engineering, but it is also conceptualized in terms of binary oppositions (for instance the one between technology and society) and its development is articulated in linear terms, as a controllable sequence of steps. This philosophico-technical conjuncture is what, in the words of Timothy Clark, Derrida understands as the 'complicity of technology with metaphysics' (Clark, 2000: 248). And yet, as the thinkers of 'orginary technicity' have shown, technology cannot be fully conceptualized within the Aristotelian framework (Beardsworth, 1996).[10] In fact, the understanding of software as a tool is continuously undone by the unexpected consequences brought about by software – which must be excluded and controlled in order for software to reach a point of stability but which at the same time remain necessary to its development.[11]

As we have seen at the beginning, the snail of *Monsters vs Aliens* is the point where the instrumentality of technology undoes itself, because technology is always *both* a monster and an alien, an instrument and a threat, a risk and a promise. This is the fundamental double valence of the unexpected as both failure *and* hope. Like Derrida's *pharmakon*, technology entails poison and remedy, danger and opportunity (Derrida, 1981). The unexpected is always implicit in technology, and the potential of technology for generating the unexpected needs to be unleashed in order for technology to function *as* technology. The attempt to control the unexpected consequences of technology is ultimately destined to fail - and yet it must be pursued for technology to exist. For this reason, every choice we make with regard to technology always implies an assumption of responsibility for the unforeseeable.

This is the problem that I have started from – namely, the fact that we constantly need to make decisions about a technology which is always, in Stiegler's words, somehow opaque. These decisions are profoundly political and they influence our very existence as human beings – not just as users of tools and machines but also as beings that co-emerge and co-evolve with technology. If one takes into account the unavoidable opacity of technology, no Habermasian way out of this dilemma can be imagined – namely, it is not enough for policy makers and citizens to make 'informed' decisions regarding technology. Of course, such decisions are inevitable and necessary, but it must also be kept in mind that not everything in technology is calculable, and that therefore every decision about technology is an assumption of responsibility for something that we cannot actually foresee. And yet a decision must be made, and responsibility needs to be taken. The more *ethical* decisions are the ones that take into account – or at least do not mask - this dilemma and that give account of their own reasons. By opening new possibilities and foreclosing others, our decisions about technology also affect our future. Thus, making responsible decisions about technology becomes part of the process of the reinvention of the political in our technicized and globalized world. Rethinking technology becomes a form of imagining our political future.

Endnotes

[1] Ostensibly the film here taps into the popular tradition of superheroes that has dominated American comic books for decades and that has subsequently crossed over into other media. The so-called 'origin stories' associated with superheroes, which explain the circumstances by which the characters acquired their exceptional abilities, often involve experiments gone wrong (see, for instance, Reynolds, 1994).

[2] Judith Halberstam has been recently constructing a 'queer' archive of 3D animated features (Halberstam, 2007), where the term 'queer' means that such features incorporate a politically subversive narrative which is cleverly disguised in a popular media form aimed at children. For instance, according to Halberstam the CGI animated film of 2003, *Finding Nemo,* depicts the title character - a motherless fish with a disabled fin – as a 'disabled hero' and links the struggle of the rejected individual to larger struggles of the dispossessed (Nemo leads a fish rebellion against the fishermen). Halberstam proposes the term 'Pixarvolt' to indicate movies depending upon Pixar technologies of animation and foregrounding the themes of revolution and transformation. For her, the Pixarvolt

films use the individual character as a gateway to stories 'of collective action, anti-capitalist critique, group bonding and alternative imaginings of community, space, embodiment and responsibility' (Halberstam, 2007: non-pag.). In a sense, it could be said that the monsters in *Monsters vs Aliens* yield themselves to a queer reading - actually, queer references seem to have become quite commonplace in animated features. For instance the Missing Link is a parody of excessive masculinity (notwithstanding his machismo and his gung-ho attitude to fight, he is comically out of shape) and has a gay bond with Insectosaurus; the monsters perform part of their first battle against Gallaxhar on a stolen San Francisco bus directed to the Castro, and, even more tellingly, the transformation of Susan into a monster frees her from all heterosexual social expectations and places her in a queer alliance within other social outcasts. Nevertheless, it is debatable whether the narrative of the film can be read as subversive, since the monsters' community seems not so much to constitute an alternative to the mainstream society as a weapon in the hands of the American government – although one could argue that such subversive narratives are at their most intriguing when they are apparently neutralized. As I will show in a moment, the neutralization of the monsters in *Monsters vs Aliens* is in fact only revealed as apparent if one focuses on their relationship with technology.

[3] In *Of Grammatology* (1976) Derrida famously shows how the incest taboo is the unthought of structural anthropology – that is, a concept that cannot be thought within the conceptual system of the discipline because it escapes its basic opposition between nature and culture. In fact, the incest taboo appears to be neither completely natural nor totally cultural, thus constituting the 'point of opacity' of structural anthropology. In turn, in Derrida's words (1980) a point of 'opacity' is a concept that escapes the foundations of the conceptual system in which it is nevertheless located and for which it remains unthinkable. For Derrida in every conceptual system we can detect a concept that is unthinkable within the conceptual structure of the system itself – therefore, it has to be excluded by the system, or, rather, it must remain unthought to allow the system to exist. A deconstructive reading looks for points of opacity – that is, for points where the system 'undoes itself'. For instance, a deconstructive reading of a specific instance of technology would therefore need to ask: what is it that has to remain unthought in order for such technology to exist?

[4] The report of the first NATO Conference on Software Engineering, held in Garmisch from 7th to 11st October 1968, was edited by Peter Naur and Brian Randell soon after the conference. NATO was in charge of the actual printing and distribution, and the report became available three months after the conference, in January 1969 (Naur & Randell, 1969). The report of the second conference, held in Rome from 27th to 31st October 1969, was edited by John Buxton and Brian Randell, and published in April 1970 (Buxton & Randell, 1970). Both reports were later republished in book form (Buxton, Naur & Randell, 1976). In 2001 Robert M. McClure made both reports available for download in pdf format at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

The pagination of the pdf version slightly differs from the original printed version. All the references made in this article are based on the original pagination.

[5] In the late 1960s the unit of measure for determining whether a system was 'large' was the number of lines of code it contained. (A large software system could include several thousands of lines of code.) Another popular unit, still used nowadays, was the 'man-year', that is the number of years an average programmer would spend on the system if he or she were to develop that system by themselves. A few years after the Garmisch conference, a sub-unit of the man-year – the man-month - became the title of the classic of Software Engineering, Frederick Brooks' *The Mythical Man-Month* (1995).

[6] Heidegger's understanding of technology is in turn deeply connected to his philosophy of time. For Heidegger (1977) modern technology is a form of calculation, and calculation has its roots in our relation to the future, and in our attempt to determine future possibilities, which we fear precisely because they appear indeterminate. Heidegger describes this process as 'anticipation' or 'concern': our attempt to control the uncertainty of the future creates the basis for calculation. Understood in a broader historical context, this is what Heidegger identifies as the turning of Western thought into calculation in the modern age. This is also why for Heidegger technology has a central role in defining modernity.

[7] HCI technologies aim at facilitating the use of computers by human beings. They presuppose a certain model of the user that has been criticized, for instance, by Matthew Fuller (2003). Fuller highlights the narrowness of the model of the user embedded in

HCI, which he understands as 'functionalist' (Fuller, 2003: 13), or represented in terms of functions performed, of tasks and efficiency. Fuller is critical of such 'idealization' of the user and argues for a shift from the model of the individualised user typical of standard HCI towards different approaches, such as Participatory Design – where users provide continuous feedback to programmers in a process of cooperative design, and in general to some more 'critical' (or even subversive) understandings of software. And yet what I want to point out in this article is that the emergence of the 'user' in the Garmisch conference report shows how the possibility of getting important feedback from the users has always been present in the theories and practices of Software Engineering, and that the 'user' is inscribed in these theories and practices not just as an 'idealization' or a 'function' of the system, but as a constitutive force within the process of software development.

[8] Linux is a Unix-like operating system started in 1991 by Linus Torvald. It is one of the most famous examples of open source software.

[9] According to Stiegler, contemporary technology is particularly difficult to understand because it has a totally new relation with time. He expresses this fact with the image of a technological device that 'goes faster than its own time'. Stiegler's favoured analogy is that of 'a supersonic device, quicker than its own sound', whose breaking of the sound barrier provokes 'a violent sonic boom, a sound shock' (Stiegler, 1998: 15).

[10] I am referring here to the alternative, non-Aristotelian tradition of thought on technology that starts with Martin Heidegger and includes Jacques Derrida and Bernard Stiegler, among others. Timothy Clark (2000) calls this the tradition of 'originary technicity' – a term he borrows from Richard Beardsworth (1996). This term assumes a paradoxical character only if one remains within the instrumental understanding of technology: if technology were instrumental, it could not be originary – that is, constitutive of the human. Thus, the concept of 'originary technicity' resists the utilitarian conception of technology. The thinkers of 'originary technicity' point out that technology is actually constitutive of philosophy, since, by providing the support for the inscription of memory, it allows for transcendence and therefore for thought.

[11] I want to emphasize at this point that these observations are not automatically valid for *all* software. Every instance of software needs

to be studied in its singularity, and problematized accordingly. What is more, the opacity of software cannot be dispelled merely through an analysis of what software 'really is' – for instance by saying that software is 'really' just hardware (Kittler, 1995). Rather, one must acknowledge that software is *always* both conceptualized according to a metaphysical framework *and* capable of escaping it – and the singular points of opacity of singular instances of software need to be brought to light.

References

Beardsworth, R. (1996) *Derrida and the Political,* New York: Routledge.

Bolter, J. D. (1984) *Turing's Man: Western Culture in the Computer Age,* London: Duckworth.

Brooks, F. P. (1995) *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Reading, MA: Addison-Wesley.

Buxton, J. N., Naur, P., & Randell, B. (eds) (1976) *Software Engineering: Concepts and Techniques,* New York: Petrocelli-Charter.

Buxton, J. N., & Randell, B. (eds) (1970) *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969,* Birmingham: NATO Science Committee.

Clark, T. (2000) 'Deconstruction and Technology', in N. Royle (ed.), *Deconstructions. A User's Guide,* Basingstoke: Palgrave, 238-257.

Derrida, J. (1976) *Of Grammatology,* Baltimore: The Johns Hopkins University Press.

Derrida, J. (1980) 'Structure, Sign, and Play in the Discourse of the Human Sciences', in *Writing and Difference,* London: Routledge, 278-294.

Derrida, J. (1981) 'Plato's Pharmacy', in *Dissemination,* Chicago, IL: University of Chicago Press, 63-171.

Derrida, J. (1985) 'Letter to a Japanese Friend', in R. Bernasconi & D. Wood (eds), *Derrida and Différance,* Warwick: Parousia Press, 1-5.

Derrida, J. & Stiegler, B. (2002) *Echographies of Television: Filmed Interviews,* Cambridge: Polity Press.

Fuller, M. (2003) *Behind the Blip: Essays on the Culture of Software,* New York: Autonomedia.

Galloway, A. (2004) *Protocol: How Control Exists after Decentralization*, Cambridge, MA: MIT Press.

Habermas, J. (1991) *The Theory of Communicative Action*, Cambridge: Polity Press.

Halberstam, J. (2007) 'Pixarvolt: Animation and Revolt', *Flow Journal* 6 (6): non-pag., http://flowtv.org/?p=739.

Hansen, M. (2004) '"Realtime Synthesis" And The Différance of The Body: Technocultural Studies In The Wake Of Deconstruction", *Culture Machine* 6: non-pag., http://www.culturemachine.net/index.php/cm/article/view/9/8.

Heidegger, M. (1977) *The Question Concerning Technology and Other Essays,* New York: Harper and Row.

Kittler, F. A. (1995) 'There Is No Software', *CTheory*: non-pag., http://www.ctheory.net/articles.aspx?id=74.

Licklider, J. C. R. (1969) 'Underestimates and Overexpectations', in A. Chayes & J. B. Wiesner (eds), *ABM: An Evaluation of the Decision to Deploy an Anti-Ballistic Missile System,* New York: Signet: 118-129.

Mackenzie, A. (2003) 'The Problem of Computer Code: Leviathan or Common Power?', non-pag., http://www.lancs.ac.uk/staff/mackenza/papers/code-leviathan.pdf.

Manovich, L. (2008) *Software Takes Command,* http://lab.softwarestudies.com/2008/11/softbook.html.

Naur, P., & Randell, B. (eds) (1969) *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch,* *Germany, 7th to 11st October 1968*, Brussels (Belgium): NATO Scientific Affairs Division.

Randell, B. (1979) 'Software Engineering in 1968', *Proceedings of the IEEE 4th International Conference on Software Engineering*, Munich: 1-10.

Randell, B. (1998) 'Memories of the NATO Software Engineering Conferences', IEEE Annals of the History of Computing 20(1): 51-54.

Raymond, E. S. (2000) 'The Cathedral and the Bazaar', *First Monday* 3(3): non-pag., http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/578/499.

Reynolds, R. (1994) *Super Heroes: A Modern Mythology*, Jackson, MS: University Press of Mississippi.

Shaw, M. (1989) 'Remembrances of a Graduate Student', *Annals of the History of Computing, Anecdotes Department,* 11 (2): 141-143.

Shaw, M. (1990) 'Prospects for an Engineering Discipline of Software', *IEEE Software*, 7 (6): 15-24.

Sommerville, I. (1995) *Software Engineering*, Harlow: Addison-Wesley.

Stiegler, B. (1998) *Technics and Time, 1: The Fault of Epimetheus*, Stanford, CA: Stanford University Press.

Stiegler, B. (2003) 'Our Ailing Educational Institutions: The Global Mnemotechnical System', *Culture Machine* 5: non-pag., http://www.culturemachine.net/index.php/cm/article/view/258/243.

Filmography

Monsters vs Aliens (2009), directed by Rob Letterman and Conrad Vernon.

The Net: The Unabomber, LSD and the Internet (2003), directed by Lutz Dammbeck.