The Naming of Systems and Software Evolvability

Martin J. Loomes, Chrystopher L. Nehaniv and Paul Wernick
School of Computer Science
University of Hertfordshire
College Lane, Hatfield, Herts AL10 9AB, United Kingdom
{M.J.Loomes, C.L.Nehaniv, P.D.Wernick}@herts.ac.uk

***Abstract*— Software systems are unlike most entities whose existence, persistence, development, and integrity as single individuals are presupposed by ordinary acts of naming. This paper broaches the issue of how naming practices in software evolution may significantly impact software maintenance and evolvability. We explore how naming in the realm of software is unlike naming of other types of phenomena to which we apply usual human naming practices. Such naming practices have been naively generalized to the realm of software. In the software realm, naming practices have been co-opted for political roles in reification as well as in the mobilization of commitment and resources.**

I. Introduction - What's in a Name?

One important issue regarding the maintenance of a system is the fairly obvious point that before we can "maintain" we must have "a system" upon which our maintenance can be deployed. But what exactly does this mean? This is a question that, we believe, has not been given the attention it deserves in the Software Engineering community, perhaps as a consequence of the often cited movement towards a Postmodern perspective where "theory" and "'philosophy" are seen as distractions from the really important activities of "getting things done".

In this paper we suggest that there is an important area of debate that is rarely (if ever?) discussed: what creates a "system" as an entity that has some existence in its own right and hence entitles it to a programme of maintenance? In particular, we will focus on one area where (as far as we are aware) there has been no discussion, *the naming of systems*. This is an extremely complex issue, and our intention here is not to present a coherent treatise on the subject, but to offer up a series of questions in the hope that philosophers, psychologists, linguists, and other experts in these sorts of fields might help us to clarify the issues. An essential general reference to the whole area of naming is Salmon [19].

Our preliminary discussions on the topic have revealed one thing: the question explodes in multiple dimensions before we can even think of entering an analytical phase of seeking clarification and "answers". Perhaps this paper will help the process along by promoting some interest. We should, however, explain briefly why these issues are of relevance to the Software Maintenance community.

II. What is Software Maintenance?

Consider the task of maintaining a motor car engine: we have a set of conditions that specify the correct set up and behaviour of the engine, and maintenance consists largely of taking the existing engine and making minor changes so that these conditions are met. Adding oil until the correct mark is reached, adjusting the timing until it is within given tolerances, and so on. We may need to replace a component, such as a filter, as a way of returning the engine to the correct state. We would be very surprised, however, if the mechanic informed us that, as part of the maintenance programme, the motor vehicle could now travel underwater, cook our evening meal, and receive 32 digital television stations. We might even be a little annoyed if informed that, in "addition", the vehicle could no longer be driven on roads.

In the software design culture, however, that is precisely what we have come to expect (and accept). As Turski notes:

> "The crux of the matter is that an ineptly chosen term masks the real issue. (Once again, sloppy linguistic habits and a childish enthusiasm for new games that can be played without rules have lead us astray.) Maintenance, as defined by dictionaries, is the act of maintaining, i.e. of keeping in an existing state, of sustaining against opposition or danger, etc. Yet, to quote a friend of mine, software engineering is the only discipline where adding a new wing to a building would be considered as a maintenance activity." [22, p. 107]

We would argue that one aspect of this problem is the way we create the impression that an artifact is of a status to be maintained in the first place. One important aspect of this is the decision to "name", for this provides identity and also something that we can relate to in political and emotional terms. In naming a system, not only do we risk the reification of something without necessarily considering all of our actions, but we also may be creating some notion of stability or persistence that was not intended.

A. Naming and Evolution of "the Word Processor"

Early "word processor" systems were designed largely as software emulators of typewriters, the latter having changed relatively little in the previous 50 years. It might be reasonable, therefore, to expect a maintenance phase as we improved our word processors so that they became better emulations. Of course, what happened is that the technical potential of such interactive software systems was quickly realized, and developments took place that moved them far from the realms

Proceedings of the 2005 IEEE International Workshop on Software Evolvability (Software-Evolvability'05)
0-7695-2460-5/05 $20.00 © 2005 IEEE

of typewriters [11]. This is progress, and not at all problematic. However, something rather more subtle accompanied this movement: various word processors were *named*[1] en route. Thus we had "Word" "Word Perfect" "WordStar" , "MacWrite", and many others. At the time, this seemed quite harmless and natural, but what are the consequences?

B. Speech Acts

Note that the naming process was carried out (to use Austin's term) by *speech acts* [1], most often by the employment of names with the tacit presupposition of the existence of "the system" – treating the system's very existence and its conceptual coherence as an entity as a *fait accompli*. Naturally, such naming was carried out in a very uncontrolled fashion.

A *speech act* can be informally defined as something said or written by somebody that in itself constitutes the act. Examples include naming a ship, promising to do something, or saying "I do" in a marriage ceremony. In order for the speech act to be effective, it must conform to certain conditions, initially described by Austin [1, pp. 14–15] and later formalized and substantially extended by Searle ([20], [21, p. 158–9]). These conditions are referred to as felicity conditions, and failures in such conditions as infelicities. Infelicities can arise in a speech act due to the correct procedure/ceremony for the speech act not being followed or being followed by somebody without the authority to perform it, e.g. the authors of this paper are unfortunately unable to pass legislation to govern the United Kingdom since they are not members of its parliament. Other infelicities can arise due to the procedure as laid down not being carried out correctly or completely. Another class of infelicity occurs when a speech act is performed insincerely, in the course of a play or other mimicking of reality, e.g. a marriage conducted by actors as part of a theatrical performance is not valid; whereas uttering an insincere promise does constitute an (abused) act of promising. [1, p. 16].

Reference via a name is explicitly treated by Searle as a speech act [20, Ch. 4]. The concepts of a speech act, and the definitions of felicity conditions and infelicities, bring into focus the question of how and when a software system can be, and is, validly named. Who can name a system? How is this naming performed?

Is there some form which is a procedure of naming, and if so how is the completeness and correctness of its performance evidenced? Can a system be named insincerely, say if a system which is not intended to actually be developed is announced to the public to spoil a competitor's marketing plans?[2] What are the implications of the naming of a system of its being cancelled after having been announced with sincere intentions?

C. Naming Acts in Practice

For software systems, such as in the naming of the word processing systems mentioned above, there were no discussions to clarify under what felicity conditions such acts were safe. It was unclear who had the authority to invoke the names (we will not be drawn here into the issues surrounding the legal positions of such names). There were no apparent rituals, such as Baptism, to name the systems. And, perhaps more worrying, no apparent understanding of the implications for the various stakeholders that naming entails. Once we had been given, e.g., "Word", however, clearly there was something in existence that was a potential candidate for maintenance, even if it was unclear what this might mean for the system in question.

Applying names, version, and release numbers to collections of software code can help clarify reference, but they do not completely solve problems that arise when names are used as if they referred to natural kinds or unambiguous species terms. Viewing the naming of systems as a social process (as suggested for other aspects of software [3], [4]), the degree to which naming can or should be formalized or regulated is a completely open question. At this stage we can only glimpse some of its the consequences of naming practices.

III. Presupposition of Individual Integrity

A. Life-Cycle Model

We should note in passing that many of the problems of system identity are inherently built in to the life-cycle model of system "development". For example, there is the usual ambiguity in the way that we view such cycles in biological sciences: is it the life of an individual that is being focused upon, or that of the development of a species? If it is one individual, say Freddy the Frog, we traditionally have some understanding of what Freddy "is" – the "encoding" of Freddy is within his inherited genetic material and he will develop in accordance with the laws of nature given a suitable environment (cf. [15]). Freddy will have some existence in the real world (embryonic Freddy is only of interest because we have faith that the "real" Freddy will soon be happily hopping around with a genuine external presence). Of course, modern biotechnology has caused us to question some of these assumptions. We would argue that using this analogy in Software Development presents even more problems, but that these are rarely recognized, let alone made explicit.

Of course, it might be argued that the life cycle metaphor in Software Engineering is intended to suggest not "Freddy the Frog" developing, but the ways in which the species "Frog" propagates and evolves. This would certainly seem more plausible, but is no less problematic. In the case of our frogs, we have some external reference as to what the species is — the life cycle thus refers to something we are able to appreciate as having an independent existence. Indeed, it was the miraculous existence of "Frogs" in general and Freddy in particular that made us seek an understanding of how life propagates and evolutionary changes occur. That was, of course, the origins of the Software Life Cycle too, but now the term seems to have become more instrumental, defining the process we should follow.

[1] We will leave aside for the moment the equally problematic issue of adopting the generic name "word processor" - cf. sec. III.B.

[2] A related area is the advent of web domain names and their subsequent creation and marketing as a commodity, as well as the use of web domain names that are misleadingly similar to well-known ones.

B. Species and Natural Kinds of Software?

Species in natural language (as opposed to biology) are defined by paradigmatic examples. Clearly by this description Windows could *not* be the name of a species in this sense, for we cannot apply the name Windows without legal permission, and its development route is important to the name (Microsoft only). If we explore this further it gets quite complex. So, what do we understand by named systems such as "Windows"? What sort of name is it? Is it a proper name that denotes some thing perhaps the master set of discs held in a safe at Microsoft Head Office? Or perhaps "Windows" is a like a *natural kind term*, such as "water", denoting something by virtue of the properties it possesses? Natural kind terms were introduced to discuss names for things in classes that, as the name suggests, occur naturally – that is, that have some independent existence. It is far from clear that we can possibly consider their use in the very artificial world of software systems, but there may be some mileage in a brief diversion into one aspect of this. Putnam [18] makes the point that terms such as "water" have been used for centuries, pre-dating much of our current understanding of the term. Indeed, the term might well have been applied to many colourless, odourless, liquids that were not H_2O at all. The crucial issue is that there should be some *sameness* relation that is used at the time to test substances against those which we all accept as water.

"The key point is that the relation *same* is a *theoretical* relation: whether or not something is the *same* liquid as this may take an indeterminate amount of scientific investigation to determine. Thus the fact that an English speaker in 1750 might have called [something with chemical composition] XYZ "water", whereas he or his successors would not have called XYZ "water" in 1800 or 1850 does not mean that the "meaning" of "water" changed for the average speaker in the interval." [18, Putnam, pp. 702–703].

We would argue that there is no agreed upon *theoretical* understanding of what constitutes *sameness* for "species" in software, such as Windows, which does not comprise a natural kind. Accepting these software names as natural kind-like terms may be a dangerous practice leading to the misuse and abuse of reference in arguments concerning the properties of such "systems".

From this perspective we might want to consider whether our systems have some existence independent of the theoretical presentations that we give them at particular instances of time. This is a counter-intuitive view for most practitioners, as it suggests that design is driven by theoretical considerations. For a more detailed discussion of this viewpoint, see [10].

If we do want to pursue this approach, however, we need to consider the implications of accepting as natural terms things which are not rigid designators. The assumption made by Putnam and others is that terms such as "water" denotes some well-defined entity. But in case of a named product, subsequent speech acts may be carried out referring to identically denoted products that slip in unexpected modifications of "the" system.

IV. Evolution, Naming and Persistence Entities

Naming of objects, natural phenomena, whether individuals or classes of them, works in a particular way that generally pre-supposes the existence and the continued persistence of the entities named. When we name persons these assumptions are naturally satisfied, and we are able to apply a name without much ambiguity to a person even though the individual changes and develops through radically different forms in the course of life. Most biological organisms have a natural persistence and integrity that allows us to do this. Our usual ways of using naming have been co-opted (or "exapted") to a new realm when we apply naming to software systems.

A. Naming Species

Biological species and software 'species' are radically different. For software, we lack both an adequate concept of species and an adequate notion of what constitutes an individual. Named individuals in our usual experience do not bifurcate and become different instances of a single entity with different characteristics and developmental routes. If Freddy the Frog were to do so, would we continue to apply a single name to it/them?

Such bifurcation however is frequent in the software realm, suggesting a vague parallel with the species concept in biology. People use names for 'species' of software that exist in multiple instances, versions, and releases. What governs the naming of species (e.g. is Windows a species, or Word, or web browsers, ..?). Are WIMP/windowing systems a species? If so, what holds them together, esp. if the analysis, design, code are very different?

In biology there are several more or less clearly defined and well-debated criteria for testing whether a given population comprises a species (cf. [12] for a discussion), but to our knowledge no such coherent species concept has been developed for software systems.

B. Evolution in Software and in Biology

Evolvability in biology (and some of its generalizations) is the capacity for generating adaptive heritable genotypic and phenotypic variation [15]. It is a striking property of biological systems that has not yet been successfully understood in detail, nor in its formal and system-theoretic aspects, nor has it been successfully modelled computationally or applied in software systems or evolutionary computation. How to achieve robustness, adaptability, and flexibility in facing changing requirements and environments is a paramount issue for software and related systems, not yet adequately addressed by previous work either in computer science or biological systems.

C. Individuals

Darwin's broad sense of evolution in organismal species as "descent with modification" applies at the level of populations of the individuals in a species over time undergoing a dynamical process with heritability, variability, selection ("struggle for existence") and limited resources. Populations rather than individuals evolve (although individuals may change and

develop in their life times), but *well-defined individuals* are required for the Darwinian theory to apply [13], [2], [14]. Persistence of changing entities is a weaker analogue of well-defined individuality in an evolutionary dynamic, and occurs also in other candidate spheres of evolutionary phenomena such as memes, software, or physical technological artifacts [16].

The software engineering community uses the term 'evolution' in a broader sense that also focuses on the descent with modification of software systems, but does not actually presuppose populations of competing individuals of the same species. Where software systems are seen as being modified and maintained in the face of changing requirements and contexts of use [4], [6], selection (success in the struggle for existence) is not usually discussed; and, increasingly, system boundaries are becoming difficult or impossible to delineate. Nevertheless, explicit empirical laws for the evolution of particular classes of software systems can be formulated [9].

Discussions of evolution of software, therefore, and of evolution in some of the other realms mentioned above (physical artifacts, memes), require a somewhat more general theory than that of Darwinian evolution with its individuals. Most importantly, software evolution differs from biological evolution in a crucial way related to naming: There is currently no well-circumscribed notion of what constitutes an individual software system.

D. Theory Building and the Evolvability of Software

Given the considerations on naming discussed here, it is abundantly clear that the existence and persistence of single entity over a longer temporal extent is not to be taken for granted when we move from the world of organisms or everyday objects into the world of software. It might even be the case that the notion of a developing individual – "the system" and "its life cycle" is not an appropriate metaphor in the realm of software maintenance and evolvability. An alternative metaphor is to developed *problem solving* and *theory building* as the central focus of software engineering, where software systems become not individuals, but instead, manifestations (in fact epiphenomena) of the more central activities of problem solving and theory building aimed at understanding and manipulating a given problem domain (cf. [10], [11]).

V. Example: Naming and Referring to Blobbo

Let us suppose that we are dealing with the names of particular entities. Consider a fictitious software enterprise. A company wishes to develop a system to handle its cleaning schedules for premises, including purchasing supplies, allocating staff, health and safety issues, and various other features that it has not yet specifically identified. We would argue that, left in this vague state, development could proceed, but at some point the individual stakeholders are likely to decide upon a name for the enterprise. Let us suppose that they decide to call it "Blobbo". One important issue arises immediately: Does "Blobbo" designate the system or the project? We would argue that whatever the intention at the outset, the move to "Blobbo" denoting the system (in some sense) is likely to occur. Thus "project Blobbo" becomes "system Blobbo". References to "Blobbo" constitute speech acts that reify Blobbo as a system. Why is this important? As a project, Blobbo is clearly something being carried out by people. As an artifact, however, there is a very real risk or hope that Blobbo will be brought into existence. It will be reified. We will talk of Blobbo as if it has some rights and life of its own. It will "develop" rather than be designed. What are the implications of the speech acts of naming and referring to Blobbo? Will stakeholders now act as if the system has been brought into existence? We now proceed as if Blobbo refers to the system. Exactly what is Blobbo? Note that, during the early stages of the project there is 'no' Blobbo. We cannot point at it, or touch it. No instrument can detect it. It is the idea of a Blobbo rather than the reality. The decision to name it (like the unborn child or yet to be conceived child) makes it more real, so that we can interact with it in subtle ways. Note that strictly, as it has no existence, it cannot "change", but people's perceptions and ideas of it can change.

Our projections of what the system might be are constantly changing as it is discussed. Here we are firmly in the world of modalities, where we need to consider the potential existence of Blobbo in a number of possible worlds. One way to consider this is to think of Blobbo as a system with state. Kripke likens this to the situation with a pair of dice: we are happy to think of the values shown on the faces of a pair of dice even if we don't know exactly what they are [5, pp. 16-21]. There has been extensive debate in philosophy about such issues, but this usually rotate around questions such as "Would the term 'Nixon' still designate the same person if he had not been President of the United States?".

For our purposes we are discussing the potential referents of a name for which no real entity yet exists. Once again there have been discussions on the naming of non-existent things like unicorns, but there is no intention that, helped by the process of naming, the unicorn will be brought into existence.

Let us move on to the situation where Blobbo has been given some substance (either as a specification or as some executable code). At this stage we need to consider what happens if Blobbo changes. There seems little concern that changing a few lines of code in Blobbo still leaves us with Blobbo. But suppose we were to decide that Blobbo should now be the name of the system that carries out all personnel functions of the enterprise, or stock control, or investment functions, ... How much change can we accept and still use the name? To quote Kripke,

> There is some vagueness here. If a chip, or molecule, of a given table had been replaced by another we would be content to say that we have the same table. But if too many chips were different, we would seem to have a different one. The same problem can, of course, arise for identity over time. Where the identity relation is vague, it may seem intransitive; a chain of apparent identities may yield an apparent

> non-identity... It seems , however, utopian to suppose that we will ever reach a level of ultimate basic particulars for which identity relations are never vague and the danger of intransitivity is eliminated. The danger usually does not arise in practice, so we ordinarily can speak simply of identity without worry. Logicians have not developed a logic of vagueness. [5, p. 51]

We would argue that, on the contrary, if Turski is to be believed, in Software Engineering this danger arises all the time. The ease with which components of software systems can be changed leads to precisely the lack of transitivity and vagueness that should cause us concern. Do we have any understanding of what a simple sounding sentence such as "you cannot do that in Windows" actually means? Or "Blobbo will be fully tested by next quarter."

VI. Commitment and Mobilization of Resources via Naming

The act of naming seems to attribute solid foundations to a system. It "is". It has some status in the world. The mere act of talking about it gives political stature and people refer to "it" in the real world as if it existed, even when it does not exist. Naming the system helps serve to organize and direct the flow of resources, including the activity of people, toward its realization (cf. [8]).

What happens when the "it", the expected system fails to become reality? Writing of the Aramis subway transport system for Paris that after many years of effort did not become a real entity and fulfill the tacit promise of its having been named, philosopher of science and technology Bruno Latour investigates what led to the project's demise; characterizing the situation, he concludes:

"Either Aramis really existed and it had been killed (the elected officials, the Budget Officer, the politicians had killed it; there really had been murder, blindness, obscurantism), or else, at the other extreme, Aramis had never existed: it had remained inconceivable since 1981, and a different crime had been committed by a different sort of blindness, another obscurantism; for years on end they'd been drawing funds for nothing – a pure loss." [7, p. 279]

Latour goes on the make the point that the project had reached a state where differing groups behaved as if Aramis were both killed and had never existed - a paradox, which he explains by observing that "Aramis" had come to mean two different things to those inside the project (such as designers and developers) and those outside (such as politicians) and that the two groups "did not discuss it. They [the politicians and other outsiders] don't know what research is. They think it amounts to throwing money out the window! Whilst everything is shifting around inside the Aramis mobile unit, outside everything is carved in stone" [7, p. 282].

Thus Aramis, at the outset, denotes a shared idea, but for one group the role of the name is to fix the concept over time, for the other it denotes the current state of a shared understanding. The recipe for disaster is firmly established - designers will report that Aramis is "developing" nicely (meaning the shared understanding is continuing to evolve), and the politicians will hear that the development is closer to their original ideas of what Aramis should be.

VII. Social Use of Names

At this stage perhaps we should consider the social use of names. There is a point of view that considers names to be simply things that refer to something in the real world. This is simplistic and clearly inappropriate for our needs, for – again to quote Kripke – "that's not what most of us do. Someone, let's say, a baby, is born; his parents call him by a certain name. They talk about him to their friends. Other people meet him. Through various sorts of talk the name is spread from link to link as if by a chain." [5, p. 91]

Thus the name and what it refers to develop through social interaction. Where the name refers to some object in the physical world, this need not be problematic, but what happens in the case of Blobbo? How do we ensure that, in the early stages, Blobbo will be a consistent being, capable of realization? How many systems, having been named, are then pushed by social discourse into a position where they simply cannot be realized?

VIII. Conclusions - Epilogue (or Prologue)

Practices of naming and reference shape how software evolution and maintenance occur. We need to decide as a community whether the issue of how we name software systems is important. And, if it is, how can we understand the mechanisms of naming practices and their impact on software systems? Might it be possible to 'reform' naming practices to achieve better software evolvability? Are there situations in the course of software evolution where we are breaking (or should be breaking) the laws and continuity of naming?

To summarize, we record some aspects of the naming of systems discussed above that are worthy of attention:

1) **Reification, Persistence, Continuity.** Naming and referring to a system tacitly asserts its individual integrity and presupposes a continual inertia of persistence over time.
2) **Commitment of resources.** As a named, reified project, a system appears to those hearing about it as having an individual integrity. Since it has a name, this suggests to others someone or some group has allocated resources for its development and is committed to nurturing it to realization and to maintaining it.
3) **Naming and no entity.** A system may be named yet nothing tangible exist, e.g. an example is an announced system. If sufficient people refer to Blobbo (even if the term designates nothing as yet), then Blobbo will develop (it is presumed).
4) **Naming and coming into being.** Via naming a project or project idea, a transition is made to 'systemhood'. The activity of the project is conflated with or converted into the system as an entity.

5) **Same name for multiple identities.** Referring to a system by name often comprises reference a collection of entities whose properties are not (and in some cases, cannot be) clearly circumscribed although they are treated like 'natural kinds' or 'species'. There are at least the following several ways in which this may occur:
   a) **Multiple copies**. This is the most trivial kind: multiple copies exist with only minor differences of parameter settings, location, etc.
   b) **Diachronic multiplicity**: multiple versions exist arising over time - descent with modification, via software maintenance and evolution.
   c) **Synchronous multiplicity**: Multiple versions exist that are all called by the same name but that may not have the same functionality, code, or manufacturer, etc.
6) **Incommensurability.** Incommensurability of naming is illustrated by the example of a transactional machine environment, intended to support both distributed databases and real-time queries. The interaction of these two requirements and the initial difficulty of reconciling them with technical issues leads the marketing department at first to sell two or more versions to different target groups of buyers. These different versions of "one" system (with a single name) come to be separately maintained, and marketing continue to sell "the system" under a single name indicating that all features will be unified in later releases. Eventually multiple, non-unifiable versions exist, still bearing the same name.

There is a myth that philosophy (as a human activity) is more relevant to day to day issues than to the deep technicalities found in areas such as software design, where simple procedures and laws will govern actions. Latour makes the point that this is not the case: "Rhetoric is the name of the discipline that has, for millennia, studied how people are made to believe and behave and taught people how to persuade others. Rhetoric is a fascinating albeit despised discipline, but it becomes still more important when debates are so exacerbated that they become scientific and technical." [8, p. 30]. In particular, those involved in rhetoric will enlist resources to help them make their points, and the subtle use of naming of artifacts is one of the tools deployed. Statements such as "we will use Object Oriented Design to implement Blobbo on a Windows platform" seek to persuade without opening the black boxes denoted by the names "Object Oriented Design", "Blobbo" and "Windows".

Naming and reference are unavoidable in software systems engineering and evolution. Names we use often remain constant while entities, properties, context, requirements, and understanding of different stakeholders are dynamic and incessantly changing. The application of naming and reference to "a system" has many consequences for software evolvability, some of which have been enumerated above, and merits further detailed attention.

REFERENCES

[1] J. L. Austin (1976). *How to Do Things with Words*, Second Edition, Oxford University Press, Oxford.

[2] L. W. Buss (1987). *The Evolution of Individuality*, Princeton University Press.

[3] J. A. Goguen (1996). Formality and Informality in Requirements Engineering, *Proc. Fourth International Conference on Requirements Engineering*, IEEE Computer Society Press, 1996.

[4] J. A. Goguen (1994), Requirements Engineering as the Reconciliation of Technical and Social Issues. In M. Jirotka and J. Goguen (Eds.), *Requirements Engineering: Social and Technical Issues*, Academic Press.

[5] S. A. Kripke (1980). *Naming and Necessity*, Oxford, Blackwell Publishers (enlarged edition).

[6] W. Lam and M. J. Loomes (1998). Requirements Evolution in the Midst of Environmental Change: A Managed Approach, *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, IEE Press, pp. 121–127.

[7] B. Latour (1996). *Aramis or the Love of Technology*, translated by C. Porter, Harvard University Press.

[8] B. Latour (1987). *Science in Action*, Harvard University Press.

[9] M. M. Lehman (1980). Programs, Life Cycles and Laws of Software Evolution, *Proceedings of the IEEE* 68(9):1060–1076,

[10] M. J. Loomes and S. Jones (1998). Requirements Engineering: A Perspective through Theory-Building, *Proc. Third International Conference on Requirements Engineering*, IEEE Computer Society Press, pp. 100–107.

[11] M. J. Loomes and C. L. Nehaniv (2001). Fact and Artifact: Reification and Drift in the History and Growth of Interactive Software Systems, *Proc. Fourth International Conference on Cognitive Technology: Instruments of Mind*, Springer Lecture Notes in Computer Science, vol. 2117, pp. 25-39.

[12] Lynn Margulis and Dorion Sagan (2003). *Acquiring Genomes: A Theory of the Origins of Species*, Basic Books.

[13] J. Maynard Smith and E. Szathmáry (1995). *The Major Transitions in Evolution*, W.H. Freeman.

[14] R. E. Michod (1999). *Darwinian Dynamics: Evolutionary Transitions in Fitness and Individuality*, Princeton University Press.

[15] C. L. Nehaniv (2003). Evolvability, *BioSystems: Journal of Biological and Information Processing Sciences* 69(2-3):77–81.

[16] C. L. Nehaniv (2000). Evolvability in Biological, Artifacts, and Software Systems. In: C. C. Maley and E. Boudreau (Eds.), *Artificial Life 7 Workshop Proceedings - Seventh International Conference on the Simulation and Synthesis of Living Systems*, Reed College, pp. 17-21.

[17] H. Putnam (1962). "It ain't necessarily so." *Journal of Philosophy* 59(22):658-671.

[18] H. Putnam (1973). Meaning and Reference, *Journal of Philosophy* 70:699-711.

[19] N. U. Salmon (1982). *Reference and Essence*, Princeton University Press.

[20] J. R. Searle (1980 [1969]) *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press.

[21] R. J. Stainton (1996). *Philosophical Perspectives on Language*, Broadview Press, Peterborough, Ontario, Canada.

[22] W. M. Turski (1981). Software Stability. *Systems Architecture: Proceedings of the Sixth ACM European Regional Conference*, Westbury House, 107–116.