Evolution in Software Systems: Foundations of the SPE Classification Scheme

Stephen Cook\(^a\), Rachel Harrison\(^a\), Meir M. Lehman\(^b\) & Paul Wernick\(^c\)

April 21, 2005

Abstract

The SPE taxonomy of evolving software systems, first proposed by Lehman in 1980, is re-examined in this work. The primary concepts of software evolution are related to generic theories of evolution, particularly Dawkins’ concept of replicator, to the hermeneutic tradition in philosophy and to Kuhn’s concept of paradigm. These concepts provide the foundations that are needed for understanding the phenomenon of software evolution and for refining the definitions of the SPE categories. In particular, this work argues that a software system should be defined as of type \(P\) if its controlling stakeholders have made a strategic decision that the system must comply with a single paradigm in its representation of domain knowledge. The proposed refinement of SPE is expected to provide a more productive basis for developing testable hypotheses and models about possible differences in the evolution of \(E\)- and \(P\)-type systems than is provided by the original scheme.

Keywords: evolution, hermeneutics, information system architecture, paradigm, philosophy of science, replicator, scientific knowledge, software evolution, software system

1 Introduction

The primary aim of this work is to contribute to the development of the theory of software evolution by re-examining and clarifying Lehman’s SPE taxonomy of evolving software systems. The SPE classification scheme has had mixed fortunes since it was first proposed in 1980. The \(E\) (Evolving) category, which includes most software systems, has been influential and become widely accepted. In contrast, the \(S\) (Specified) and \(P\) (Problem) categories have not been studied in detail, and the taxonomy’s rationale has received little attention. It has been known from empirical studies, e.g. [1, 2, 3, 4], that software systems are not uniform in their patterns of evolution. However, studies of such differences have made little use of either theory-based classifications such as SPE, or classifications based on observed properties of systems such as development process, application domain, etc. The juxtaposition of these developments raises some interesting questions. Do the same general principles of evolution apply to all software systems? Can the ‘Laws of Software Evolution’, which were based on empirical data obtained from \(E\)-type systems, be modified or extended to apply more generally, or even universally? What would constitute a sound basis for classifying evolving software systems?

To answer these questions, it seems reasonable to start from the existing SPE scheme, given the success of its \(E\) category and the absence of strong competitors. This work begins the process of re-examining SPE by describing recent progress in the following inter-related areas. We explain how the theory of software evolution can be related to generic theories

---

\(^a\) Applied Software Engineering Research Group, School of Systems Engineering, University of Reading, UK

\(^b\) School of Computing Science, Middlesex University, UK

\(^c\) Centre for Empirical Software Process Research, Department of Computer Science, University of Hertfordshire, UK
of evolution. We also show how software evolution theory provides a bridge between the technological concerns of software engineering and philosophical concepts of hermeneutics and paradigm. Our proposed unification of these concepts provides a better understanding of why the use of software systems in the real world leads to uncertain outcomes. This material is then used to propose a refined definition and rationale for the SPE categories, referred to as SPE+. Thus this work is focussed on establishing the conceptual basis of SPE+, as a necessary precursor to empirical studies of the classification of evolving software systems.

It will be apparent that the scope of this work is rather broad and touches on topics that may be unfamiliar to some readers. Some of the material may seem, at least initially, rather distant from the usual concerns of software engineering. Nevertheless, this work is based on the position that the effects of software evolution cannot be managed successfully unless a better understanding of software evolution becomes an integral part of the software engineering paradigm. The evolution of software systems cannot be fully understood solely in terms of the operations of computers and programs. To achieve a more complete understanding of software evolution requires some knowledge of developments and discoveries that have been made in various branches of philosophy and in the study of generic theories of evolution. To assist readers who may wish to explore these issues in greater depth, the bibliography is both longer and broader than usual.

The work is organised in the following way. Section 2 introduces background material and concepts. Section 2.1 discusses the concept of evolution in the context of software systems. The original SPE scheme is summarised in section 2.1.4. Section 2.2 explains the related concepts of stakeholder, architecture and global software process. Section 2.3 introduces concepts from the hermeneutic tradition in philosophy and from the philosophy of science. Section 2.4 applies these concepts to the process of requirements analysis. Section 3 explains the details of SPE+ and shows how the definitions of the categories are based on the concepts described in section 2. Finally, section 4 discusses the impact of these proposals on the theory of software evolution and suggests some worthwhile directions for future research.

2 Background and Related Work

2.1 Software Evolution

This section discusses the concept of evolution in software systems. Sections 2.1.1 and 2.1.2 show how generic concepts of evolution, particularly Dawkins’ concept of replicator, can be applied to software. Section 2.1.3 returns to the specific characteristics of software evolution with a brief summary of “Lehman’s Laws”. Section 2.1.4 summarises the original formulation of the SPE taxonomy.

2.1.1 What is evolution?

Evolution is an elusive term to define. Common-sense and dictionary definitions imply that it refers to ‘a gradual process of change and development’. This leaves plenty of room for interpretation. For example, users and administrators of databases could have different ideas about what kinds of change qualify as ‘evolution’ in a database. To the users of a database, evolution might mean that the uses of the database system or the semantics of its data have changed over time. On the other hand, database administrators might consider such changes to be within the normal use of the system. They might use the term ‘evolution’ to refer to changes in the definition of a database’s schema or the features of its DBMS, while the system’s users might be unaware of such changes or unconcerned about them.
Evolution can also be defined in ways that are independent of subjective viewpoints. A ‘top-down’ approach describes the generalised character of evolutionary processes. For example, at a recent workshop\(^1\), Lehman proposed the following very general statement, defining evolution as

‘... process of discrete, progressive, change over time in the characteristics, attributes, [or] properties of some material or abstract, natural or artificial, entity or system or of a sequence of these [changes]’

This definition captures important characteristics of evolution in many situations, including software systems. It is applicable to both natural and artificial systems, and to abstractions such as ideas. It provides a very general, universal definition of evolution that can be specialised for particular domains, such as software, natural languages, and genes.

An alternative, complementary approach works in the opposite direction, i.e. ‘bottom-up’. Such definitions focus on identifying the minimum starting conditions for evolution. For example, Dawkins\(^5\) defines evolution as ‘the external and visible manifestation of the differential survival of alternative replicators’. Blackmore\(^6\) paraphrases this as

‘If there is a replicator that makes imperfect copies of itself only some of which survive, then evolution simply must happen’. [emphasis in original]

A replicator, as defined by Dawkins\(^5\), is anything that can be copied. Genes are replicators and so are many other things. In a software context, replicators include fragments of source code, complete programs, designs, design patterns, algorithms, operating manuals, policy statements etc. Copies of replicators may be ‘imperfect’, in the sense of ‘variant’ or ‘with alterations’. This may happen accidentally, as in the case of random mutations in genes. Alterations to replicators may also happen through deliberate actions, as when a programmer adapts the source code of a program or replaces the algorithm or design pattern that is used in it.

At least in the software domain, the ‘top-down’ and ‘bottom-up’ definitions of evolution described above are consistent with each other. Whenever a process involving a software system satisfies Lehman’s ‘top-down’ definition, there will be differential survival among the replicators within the system, i.e. this kind of change in a software system always involves adding, deleting or changing one or more replicators. Similarly, from the ‘bottom-up’ perspective, whenever a process of differential survival among a collection of software replicators is sustained for a sufficient length of time, it will produce system effects that satisfy the ‘top-down’ definition of evolution.

2.1.2 A replicator perspective on software evolution

This section explores the replicator concept in more detail by illustrating some of its applications to the software domain.

In Dawkins’ model, replicators travel around in vehicles. A vehicle can be

‘any relatively discrete entity ... which houses replicators, and which can be regarded as a machine programmed to preserve and propagate the replicators that ride inside it’\(^5\), p. 302].

Genes, for example, travel around in living things, which tend to preserve and propagate them, often in very complex and elaborate ways. Software-related replicators travel around in software itself, and also in books, websites, system documents and the brains of programmers and software engineers.\(^2\)

---


\(^2\) This implies that the replicators found in software could be treated as a class of ‘meme’\(^6, 7\). Blackmore\(^6\) briefly mentions the possibility of developing memetic explanations for software evolution.
Evolution ‘takes off’ as a process when at least some replicators are germ-line, rather than dead-end, but only some of their vehicles survive long enough to propagate the replicators that are travelling in them. A germ-line replicator is ‘a replicator that is potentially the ancestor of an indefinitely long line of descendent replicators’ [5], whereas the dead-end category lack this capability. Dawkins uses the metaphor that evolution occurs when replicators aspire to immortality but some fail to achieve it sooner than others.

The approach of Dawkins and his colleagues and successors to the definition of evolution is explicitly Darwinian but their concepts are not defined in exclusively biological terms. Blackmore [6] and Plotkin [9], for example, show how this approach can be applied in non-genetic domains. The use of Darwinian concepts in theories of generic evolution is particularly interesting and relevant to the software domain. Previous attempts to make direct analogies between evolution of living things and in software systems have often been unsatisfactory. Establishing a relationship between the theories and concepts of software evolution and generic evolution seems more promising.

For example, the concepts of germ-line and dead-end replicator can be applied to software. All replicators that travel in open-source software are germ-line. This is because each copy of an open-source program can spawn new lines of indefinitely long descent for the replicators that it hosts. In principle, every copy of an open-source program could do this independently, but most do not. The situation with proprietary software products is more complex and the replicator concept can be used to explain this. The replicators in the design, source code, etc. are, in general, germ-line only within the relevant development community. They can be reused, possibly with alterations, in subsequent releases of the product or be copied into other vehicles, e.g. programs, UML diagrams, CASE repositories. However, those replicators that get copied into each end-use copy of a proprietary product are effectively dead-end. They cannot become ancestors of descendent replicators without unlicensed reverse engineering. In the absence of cooperation from the product’s owner, this process may be difficult, unreliable and exposed to the risk of sanctions.

However, if a proprietary product has been conceived, designed and implemented as a reusable and adaptable component, it can become a germ-line replicator in its own right, even if none of its source code is available as a replicator. When a piece of software is reused as a component, it takes on a replicator role and it uses the systems in which it has been incorporated as its vehicle. The component may get adapted if its interface permits this, or it may be discarded as its host system evolves. Meanwhile, the component will continue to play the vehicle role for its own ‘payload’ of replicators.

The concepts of generic evolution that have been developed by Dawkins, Plotkin, Blackmore and others provide a framework for understanding the features of evolution that are common to different domains, including software. They also provide a vocabulary for discussing the distinctive features of software evolution. For example, because software can be structured in hierarchical, recursive and reflective ways, many software artifacts can act as either replicator or vehicle or play both roles simultaneously. This can be contrasted with biological systems, where in general an entity may be either a replicator, e.g. a gene, or a vehicle, e.g. an organism, but not both at the same time.

Nested replicator-vehicle relationships in software can be very simple or arbitrarily complex. A relatively simple example is found in the pipe-and-filter [11] architectural style. Each filter is a vehicle for a collection of replicators — source code, algorithms, design patterns, etc. — and also behaves as a replicator that can be copied from one pipeline to another. More complex examples of nested replicator-vehicle relationships can be found in, for example, the use of application frameworks [12] to guide the evolution of a software system. Relatively simple kinds of similar relationships are also found in other engineering

---

3 The concept of immortality has been applied to software by Edwards and Millea [8].

4 This phenomenon has also been investigated by Lehman and Ramil [10].
structures. For example, Alexander [13] describes the hierarchical arrangement of reusable design patterns, i.e. replicators, that are involved in designing the ‘built environment’. However, some complex arrangements of replicators and vehicles, such as those found in reflective meta-programming, are only possible in software systems.

2.1.3 Theories of software evolution

The concept of software evolution can be traced back to Lehman’s 1969 study [14] of the programming process within IBM. He identified several long-term trends in software systems that seemed to be independent of the intentions of any of a system’s stakeholders (§2.2.1), e.g. programmers, project managers, marketing departments, user organisations. These trends included tendencies for programs to steadily increase in size and complexity and to become progressively harder to adapt. Initially, Lehman and Belady [15] called these phenomena ‘program growth dynamics’ but later they coined the term software evolution [16]. Since then, a growing body of research and experience has confirmed many of their original insights and contributed new information, hypotheses and investigative techniques. Lehman and Ramil [17] provide a convenient summary of the principal advances.

It is helpful to distinguish two broad approaches to the study of software evolution:

explanatory : concerned with understanding causes, processes and effects

This approach attempts to achieve a holistic view and considers, for example, the impact of software evolution on the effectiveness of organisations and the planning of organisational change.

process improvement : concerned with the development of better methods and tools

This approach addresses such questions as ‘How should software engineering activities such as design, maintenance [18, 19], refactoring [20], reengineering etc. be used to manage the effects of software evolution?’.

Lehman et al. [21] have described these complementary strands as the What? and How? of software evolution. The distinction is important because of the tendency in software engineering practice to over-emphasise short-term ‘fixes’ for fundamental problems. To surmount the limitations of ad hoc solutions, it is essential to develop process improvement techniques that are robust. This requires a sound understanding of the phenomena that the techniques address. The research described in this work falls in the ‘explanatory’ category.

Laws of software evolution

Lehman’s ‘laws of software evolution’ are a major contribution to identifying the causes and processes of this complex phenomenon. The eight laws that have been discovered so far are summarised in Table 1, adapted from Lehman et al. [22]. They describe a set of general principles for the evolution of E-type (§2.1.4) software systems.

Lehman’s use of the term ‘law’ in the context of software evolution has sometimes been misunderstood. Unlike some laws found in sciences such as physics, Lehman’s laws do not specify precise invariant mathematical relationships between directly observable quantities, and were never intended to. Their purpose is to capture knowledge about the common features of frequently observed behaviour in evolving software systems. As this knowledge deepens and becomes more detailed and reliable, it is likely that future versions of the laws may be expressed in more precisely quantified terms.

Thus law is being used by Lehman in the same sense that social scientists use the term to describe general principles that are believed to apply to some class of social situation.
<table>
<thead>
<tr>
<th>Name Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I</strong> Continuing Change [1974] <strong>E-type</strong> systems must be continually adapted else they become progressively less satisfactory.</td>
</tr>
<tr>
<td><strong>II</strong> Increasing Complexity [1974] As an <strong>E-type</strong> system evolves, its complexity increases unless work is done to maintain or reduce it.</td>
</tr>
<tr>
<td><strong>III</strong> Self Regulation [1974] The evolution process of <strong>E-type</strong> systems is self regulating, with a distribution of product and process measures over time that is close to normal.</td>
</tr>
<tr>
<td><strong>IV</strong> Conservation of Organisational Stability [1980] The average effective global activity rate in an evolving <strong>E-type</strong> system is invariant over a product’s lifetime.</td>
</tr>
<tr>
<td><strong>V</strong> Conservation of Familiarity [1980] During the active life of an evolving <strong>E-type</strong> system, the average content of successive releases is invariant.</td>
</tr>
<tr>
<td><strong>VI</strong> Continuing Growth [1980] The functional content of an <strong>E-type</strong> system must be continually increased to maintain user satisfaction with the system over its lifetime.</td>
</tr>
<tr>
<td><strong>VII</strong> Declining Quality [1996] Stakeholders will perceive an <strong>E-type</strong> system to have declining quality unless it is rigorously maintained and adapted to its changing operational environment.</td>
</tr>
<tr>
<td><strong>VIII</strong> Feedback System [1974–1996] The evolution processes in <strong>E-type</strong> systems constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable baseline.</td>
</tr>
</tbody>
</table>

Table 1: Laws of software evolution, adapted from Lehman et al. [22]

‘other things being equal, which they rarely are’. For example, Say’s Law in economics describes a general principle about the relationship between demand and supply, which may need to be modified when it is applied to particular situations. Since the theory of software evolution is similarly describing social situations that are extremely variable in practice, this use of the term ‘law’ is appropriate.

### 2.1.4 The SPE classification scheme

Lehman devised his SPE taxonomy [24] to explain why programs vary in their evolutionary characteristics. He realised that, from the perspective of software evolution, there is a fundamental distinction between programs written to satisfy a fixed and pre-existing specification, and programs developed to satisfy some need in the real world.

This insight was refined into the three types described by the SPE taxonomy. The ‘specification-based’ programs became the **S** (for Specification) type and the ‘real world’ programs inspired the **E** (for Evolving) type. A third type, **P** for Problem, was also identified. However, early studies of **P-type** programs suggested that, in practice, they always satisfied the definition of either **S-type** or **E-type**. Thus in his subsequent work Lehman ignored type **P**. A major contribution of this work is to provide a revised definition

---

5 Say’s Law can be expressed informally as ‘supply creates its own demand’. Economists, e.g. [23], have discussed different interpretations and applications of this general principle.
and description of the \( P \) type that is both conceptually sound and relevant to software engineering practice.

The notion of \( E \)-type software has achieved widespread acceptance. It has informed all of Lehman’s subsequent work and has been accepted by many other researchers in software evolution. However, the \( S \) and \( P \) categories and the rationale of the taxonomy have received less attention.

**Type S** — Programs with the following characteristics belong to type \( S \):

- all the program properties, functional and non-functional, that matter to its stakeholders have been completely defined in a specification, which in practice will be expressed in a formal language, and
- the only criterion of the program’s acceptability to its stakeholders is satisfaction of the specification [25].

These properties define conditions in which software evolution does not occur.

Once an \( S \)-type program satisfies its specification, and hence its stakeholders, it can be put to use. There is no good reason for changing it subsequently. The program cannot be improved since, by definition, it already completely satisfies its acceptance criteria. On the other hand, any change to the program exposes it to the risk that it will no longer satisfy its specification and will have to be repaired. So any change to the program will waste resources.

If a specification is changed, then in general it will be necessary to amend any program derived from it, to restore stakeholders’ satisfaction with the program. However, the definition of type \( S \) precludes this because the completeness property implies that the specification and any programs derived from it are conceptually static. If the text of an \( S \)-type specification \( Z \) — which by definition is complete — is reused in a different specification \( Y \), then \( Y \) is conceptually a new specification. It follows that \( Y \) must be implemented by a new program, although in practice this may well involve copying some replicators from previous programs. Conversely, if stakeholders treat a specification \( X' \) as an evolution of an earlier specification \( X \), then regardless of whether the text of \( X \) was reused in \( X' \), it follows that \( X \) was incomplete and therefore not \( S \)-type.

The effect of these conditions is that \( S \)-type programs are rare in the real world. Although many programs are intended to satisfy formal specifications, this is insufficient to qualify them as \( S \)-type and in general they will evolve in the manner of \( E \)-type or \( P \)-type programs. In practice, a ‘frozen’ specification rarely leads to a satisfactory system. This is because stakeholders’ satisfaction with a software system often depends on issues that are very difficult to specify completely without some experience of using the system, for example:

- programs are rarely used in isolation but need to be compatible with other software, e.g. an operating system, and a hardware platform
- many non-functional properties, e.g. usability, depend on assessments that are subjective or situation-specific
- when writing the specification, stakeholders may have included incorrect assumptions or omitted important assumptions about the application domain or the operating environment of the system

Nevertheless, despite being rarely observed, the \( S \) category is conceptually important because it defines conditions under which software evolution does not occur. The fact that these conditions are rarely satisfied has implications for the ‘global software process’ (§2.2.3) of almost all software systems. It is also important because many approaches
that attempt to increase the formality of software engineering implicitly assume that the system will not evolve, i.e. S-type conditions are tacitly assumed. Such approaches ignore the temporal dimension and their formalisms do not provide any means of representing the possible evolution of a system. For example, the Acme [26] architecture description language (ADL) [27] has no constructs to represent the situation where a system’s architecture could be different at times $t_0$ and $t_1$. Given that the IEEE definition of architecture refers explicitly to ‘evolution’ (§2.2.2), it is important to be aware of this limitation in Acme and similar ADLs. The existence of the S category within SPE makes it easier to uncover these assumptions.

**Type P** — In Lehman’s original treatment of SPE, type $P$ was derived from the observation that designing a useful, problem-solving program generally requires compromises between stakeholders’ goals. For example, tradeoffs may be made between design elegance and the need to produce practical results. In many cases, the inputs and outputs of a program can only be accurate to some level of precision, rather than correct in terms of a formal proof as type $S$ requires. This issue potentially arises in every numerical problem, other than arithmetic with integers and rational numbers.

However, Lehman did not identify any necessary characteristics of $P$-type programs and this contributed to his perception that the category was redundant. Section 3.2 proposes a revised definition that provides a justification for the category and gives it a vital role in SPE+.

**Type E** — Programs that depend on or interact with the real world belong to type $E$. They include programs that ‘mechanise a human or societal activity’ [24], or make assumptions about the real world, or interact with the real world by providing or requiring services. In general, such programs must be adapted to match any changes in the real world that affect whether the program satisfies its stakeholders’ objectives. Since the real world is dynamic, an $E$-type program must in practice be continually adapted to remain faithful to its application domain, compatible with its operating environment, and relevant to its stakeholders’ goals [28] and expectations.

Situations that include $E$-type programs can become very complex. This happens because

> ‘the installation of the program together with its associated system . . . changes
> the very nature of the problem to be solved. The program has become a part of
> the world it models, it is embedded [and executed] in it. Conceptually at least
> the program as a model contains elements that model itself, the consequences
> of its execution.’ [24, p. 1063] [emphasis in original]

An important consequence is that evolution processes in $E$-type software systems are subject to positive feedback loops. In particular, the introduction of a new or improved system may produce unexpected side-effects rather than restore equilibrium. That is to say, regardless of whether a system change satisfies the requirements of any stakeholder, introducing the change may create or expose issues that must be addressed by making further changes to the system. For example, a system change may stimulate some stakeholders to revise their ideas about the problem that they want the system to address, or the service that the system provides, or the way that the system achieves its results. Chatters et al. [29] describe a simulation of this process.

The stakeholders who experience the, possibly unexpected, impacts of system changes may be the same stakeholders who originally requested the change or other stakeholders. In either case, earlier compromises between the concerns of different stakeholders may be disrupted in unpredictable ways by ‘improvements’ to the system. Consequently, the dynamic behaviour of the global software process (§2.2.3) for an $E$-type system will be complex, difficult to predict and sometimes counter-intuitive [29].
2.2 Stakeholders, Architecture and Software Evolution

2.2.1 Stakeholders and software systems

Software systems vary considerably in the complexity of the roles that are involved in their development and subsequent use. At one extreme is the solitary programmer who writes a program solely for personal use. At the opposite extreme, many different individuals, groups and organisations can be involved in and affected by a software system over its lifetime. Their objectives, viewpoints and concerns will often differ and tend to reflect their role — e.g. customer, user, architect, programmer — in relation to the system.

The concept of stakeholder is useful for capturing the active, directed character of roles in systems. It is borrowed from management theory, where a stakeholder is

‘any individual or group who can affect or is affected by the actions, decisions, policies, practices or goals of the organization’ [30].

In the context of software architecture, IEEE defines a system stakeholder as

‘an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system’ [31].

Although different stakeholders may agree about the objectives of a system, they will usually have different concerns about it. In the context of software systems, IEEE defines concerns as

‘those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.’ [31]

For example, users tend to have concerns about a system’s functionality and usability, whereas customers may be more concerned about costs of ownership, and software engineers are likely to be concerned about maintainability and evolvability.

Stakeholders’ concerns are an important driver in the definition of architectural viewpoints. The same IEEE standard defines a viewpoint as:

‘A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.’ [31]

So a stakeholder who has concerns about, say, system evolvability, may define or reuse an existing viewpoint which abstracts the features of the evolvability quality which are deemed to be important and explains how they should be observed and represented. The viewpoint may then be used to generate evolvability views of specific systems.

The concepts of concerns and viewpoints help to explain why different stakeholders may see apparently contradictory views of the same system, as in the example described in §2.1.1. Some apparent discrepancies can be resolved by distinguishing carefully between the system in itself and a particular stakeholder’s partial view of and knowledge about the system. However, some proposed definitions of software evolution implicitly entangle these two aspects. For example, Chapin et al. have proposed a comprehensive taxonomy of software maintenance and evolution that defines software evolution in terms of ‘customer-experienced functionality or properties [of software]’ [18].

Nevertheless, differences between stakeholders’ views cannot always be reconciled by references to objective facts. Because stakeholders differ in their concerns and viewpoints,
they legitimately interpret the world in different ways, which may not be obviously commensurable\(^6\). The hermeneutics tradition in philosophy studies the process of interpretation. Some of its conclusions are discussed in section 2.3 and applied to the requirements analysis process in section 2.4.

### 2.2.2 Evolution and system architecture

This section considers the relationship between the concepts of architecture and evolution. In the context of software systems, IEEE Standard 1471-2000 provides a definition of architecture that explicitly refers to evolution:

> ‘The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.’ \([31]\)

Thus every system has architectural properties, which may be deliberate or accidental. In either case, they crystallise assumptions about the expected evolution of the system. However, the evolution that actually occurs may not be what the designers of a system’s architecture were expecting at the time when architectural choices were made. Whenever a system’s architecture incorporates assumptions about the real world that no longer hold and the discrepancy cannot be overlooked, then the system’s stakeholders may be faced with either replacing or re-architecting the system. If a software system models a real-world domain, there will always be a risk that this situation could arise.

Real-world domains have, in general, an unbounded number of properties. For example, the properties of the ‘retailing’ domain cannot be listed exhaustively. Consequently, modelling such domains in a finite software system involves an unbounded number of assumptions. Many, perhaps even the overwhelming majority, of these assumptions will be irrelevant to a particular software system at any moment. However, over time the relevance and accuracy of real-world assumptions will change in unpredictable ways. In many cases, even stakeholders who are domain experts cannot fully justify their assumptions about a domain and are obliged to infer from their past experience, which may not be a reliable predictor of the future. In other cases, stakeholders are simply unaware of assumptions that they have made. Consequently, many software systems have properties that are effectively hidden because they are not currently referenced by any concerns of that system’s stakeholders. Some of these hidden properties may be architectural, i.e. they are part of the ‘fundamental organisation’ of the system and therefore cannot be changed easily.

The relationship between architecture and evolution has also been explored in architecture’s original domain, the ‘built environment’. Alexander’s work on the role of design patterns \([13]\) in architecture has been influential in many domains, including software engineering \([32]\). One of his themes is that the architectural process, including the use of patterns, should support the gradual, piecemeal evolution of the built environment so that it becomes increasingly congruent with the changing ways in which people want to use buildings \([33]\), as opposed to more rigid approaches to design in which people have to adapt to the preconceptions of an architect. These ideas have also been explored by Brand, who identified a number of different temporal patterns in the co-evolution of buildings and their uses \([34]\).

Architectural concerns in software systems can be described in various ways \([35]\). From a software evolution perspective, Zachman’s \([36, 37]\) taxonomy of architecture viewpoints is useful. He identified five levels of abstraction and six categories of concern in architectural descriptions. Their product gives a matrix of 30 viewpoints, which Zachman proposed
as atomic components that could be combined into more complex, stakeholder-specific viewpoints. Zachman’s levels of abstraction are summarised in Table 2, where they are illustrated by viewpoints based on data-oriented concerns\(^7\). The viewpoints use different models to describe the information that is relevant to stakeholders’ concerns at each level of architectural abstraction.

---

**Contextual** : planner’s viewpoints, concerned with a system’s scope and relationship to other systems and policies.

*Example:* develop a business analysis or scenario\(^a\) that explains why some category of information is important to a system’s stakeholders.

**Conceptual** : owner’s viewpoints, concerned with a system’s fitness-for-purpose in relation to some social or business process.

*Example:* describe the organisational roles, entities, relationships and rules that are involved in creating, updating and using a particular kind of information.

**Logical** : designer’s viewpoints, concerned with the specification of the computational entities, relationships, processes, algorithms etc., and with resolving design constraints independently of any particular implementation language or product.

*Example:* define a logical data schema that specifies entities, attributes and constraints in terms of some computational data model, e.g. relational, object-oriented, deductive.

**Physical** : builder’s viewpoints, concerned with resolving construction constraints and with the impact of the engineering properties of specific technologies on a system.

*Example:* define tables, indexes, procedures etc. in terms of a specific data manipulation language, e.g. a proprietary dialect of SQL.

**Components** : assembler’s viewpoints, concerned with the physical production and assembly of a system’s components.

*Example:* construct a schedule of machine and file addresses where the database components will be located.

---

\(^a\)For a practitioner’s guide to the role of business scenarios, see ‘Part IV Resources’ of The Open Group Architecture Framework (TOGAF), [http://www.opengroup.org/architecture/togaf8-doc/arch/](http://www.opengroup.org/architecture/togaf8-doc/arch/).

---

Table 2: Levels of abstraction in architectural descriptions (after Zachman), as applied to Data concerns

The two most abstract levels, Contextual and Conceptual, share the property that they are primarily concerned with application domain-dependent information, i.e. concerns at these levels can only be fully understood in relation to the application of the system to some real-world problem. Conversely, the two least abstract levels, Physical and Components, are largely context-free with respect to application domains. An example from the Physical level is that a relational database product can be used in a wide range of application domains and its properties, e.g. locking strategy, index structure, are derived from the domain of database technology, not from any specific application domain.

\(^7\)Zachman’s taxonomy also includes Function, Network, People, Time and Motivation categories of concerns.
2.2.3 Stakeholders, system evolution and the global software process

Traditionally, software development methodologies have concentrated on two roles that people can play in the software process, ‘user’ and ‘developer’. One of the benefits of software evolution research has been to broaden this perspective. The term global software process [38] has been proposed to refer to a holistic concept of the organisational processes, roles and forces that affect the evolution of software systems. In this context, the term ‘global’ does not necessarily imply ‘world-wide’; the geographical extent of a global software process could be very small.

The actors in a global software process include the stakeholders who can make decisions that cause the system to evolve, those who carry out the changes, and those who are affected by its evolution. Although the power to implement changes to a software system might be confined to professional IT staff, decisions about its semantics and policies are usually more diffused. For example, the stakeholders of most business-related software systems will include a variety of governmental and other regulatory bodies whose decisions may invalidate a system’s assumptions and thus cause it to either evolve or become less useful [39].

Previous work by Lehman et al., e.g. [29, 22], has identified the complex role of feedback in software evolution processes. Where a software system solves problems or provides services in the real world, stakeholders usually need feedback from different parts of the global software process to help them refine their requirements for the system. The scenario that is familiar to every software developer is that stakeholders’ ideas may change while the system is being designed and built. To a greater or lesser extent, this risk can be mitigated by various methodologies for iterative and incremental software development. However, stakeholders’ ideas can also change as a consequence of using the software, or of observing its effects on the real world, or in response to any other events in the world. These effects are much harder to predict but may be more influential in determining stakeholders’ overall satisfaction with a system.

Ideally, the evolution of large-scale software systems should be managed through a defined process, sometimes referred to as IT governance

8. In practice software evolution also happens in unplanned ways. Consider a situation where the users of a sub-system decide to change their use of a data field, e.g. changing the basis for calculating depreciation values in an accounting system. If the revised use is consistent with the field’s syntax and functions, then the change may be functionally transparent to other sub-systems that share this data but still have a significant semantic effect on stakeholders who need to use the data, but may not have been involved in the change decision or even be aware of it.

Management information systems and data warehouses that import data from loosely-coupled tributary systems are particularly vulnerable to this kind of rippling evolution. Creators and distributors of information are often unaware of how semantic changes could affect indirect consumers of the information. Similarly, consumers may be unaware of semantic changes in the information that they use until discrepancies or inconsistencies come to their attention. This kind of ‘misunderstanding’ arises very easily, even within a single organisation, when information collected for one purpose, e.g. maintaining an asset register and depreciation accounts, is reused for a secondary purpose, e.g. project management. Mergers and acquisitions between organisations often create similar difficulties.

2.3 Hermeneutics and Software Evolution

We now explore the philosophical foundations of the theory of software evolution, particularly the hermeneutics tradition in philosophy and Kuhn’s extended concept of paradigm

8 See, for example, the IT Governance Institute http://www.itgi.org/
in the philosophy of science. This apparent digression from software engineering provides concepts that will be used in sections 3.2.1 to 3.2.4 to justify SPE+.

2.3.1 Software systems and the interpretation problem

It is possible to design and use software in an entirely abstract way that merely manipulates mathematical and logical expressions. However, for a software system to do something useful in the real world, its inputs, algorithms and outputs must be assigned additional meanings by relating them to the real world.

In simple scenarios, e.g. using a pocket calculator, the meaning of each computation is known only to the machine’s user and is not represented within the machine. However, most software derives its power, usefulness and complexity from more complicated scenarios where the real-world application of computations is represented within the software to a greater or lesser extent. Furthermore, when a software system is used as a control system, then it not only holds a representation of part of the real world but is also an active participant whose operations directly change the real world. Such uses of software systems involve several interpretative processes:

1. The requirements of the software must be formulated, based on some prior understanding of the problem that it will solve or the knowledge domain that it will model.
2. The requirements must be understood, related to relevant technologies and implemented in a system.
3. The results of using the software system must be understood and related to the relevant aspects of the real world.

The interpretation problem becomes even more complex when feedback loops arise from the use of a system [40, 38]. Such feedback may change the knowledge or invalidate the assumptions that had supported previous interpretations. Thus, interpretative processes continually introduce uncertainty into E- and P-type software systems, during both their initial development and their subsequent evolution. The philosophical tradition that is most relevant to understanding the interpretation process is hermeneutics, which is discussed in the following section.

2.3.2 Hermeneutics, language and dialogue

The central concern of hermeneutics is interpretation: how do readers and listeners discover meanings in texts, utterances and similar acts of communication? Habermas has defined hermeneutics as

‘the art of understanding the meaning of linguistic communication and, in the case of disrupted communication, of making it understandable.’ [41]

Although the origins of hermeneutics can be traced back to ancient Greece, the modern study of hermeneutics is usually attributed to the work of Friedrich Schleiermacher (1768–1834) and Wilhelm Dilthey (1833–1911). Their approaches tended to be positivist, i.e. they assumed that each text had a single, correct meaning that could be extracted reliably by following the right method. Mallery et al. [42] refer to this tradition as methodological hermeneutics

However, since the work of Heidegger and his successors, particularly Gadamer and Habermas, it is generally accepted that, at least in principle, multiple interpretations of a text are always possible because both the author and the reader contribute their unique experience and perspective to the interpretative process. Mallery et al. [42] refer to this tradition as phenomenological hermeneutics. Philosophers within this tradition continue
to disagree about whether and how a reader can gain awareness and understanding of their own and the author’s subjectivity. Habermas and others have stressed the role of dialogue in helping both participants, the sender and the receiver, to become aware of their own and the other’s assumptions.

The insights of hermeneutics help to explain some of the strengths and limitations of formal languages for specifying requirements. Languages such as Ross’s Structured Analysis [43] made important advances in understanding the linguistic basis of stakeholders’ requirements and encouraged the recognition of different stakeholders’ viewpoints. However, when applied to the real world, such languages must still rely on shared assumptions that cannot be enumerated exhaustively. They may also risk over-abstracting from statements that are authentic but resist formalisation, e.g. fuzzy but important concepts such as ‘user-friendly’.

Thus a hermeneutic perspective implies that some expectations of formal specification have been over-ambitious. In hermeneutic terms, some advocates of formal languages appear to have overlooked the insight that the interpretation of a text also depends on context — the situations in which language is used — and pragmatics [42] — the meaning inferred from using a sign or symbol in a context — as well as on syntax, grammar and semantics. Some more recent contributions to the formalisation of requirements, e.g. Goguen’s Algebraic Semiotics [44], have tried to avoid these traps by advocating the use of formality to assist reasoning about social situations [45], rather than to eliminate uncertainty in interpretation.

In practice, there are some domains where a methodological hermeneutics can often produce effective results, and the complexities of phenomenological approaches can therefore be set aside. These domains have been studied by Kuhn. They are characteristic of well-established scientific disciplines, such as chemistry and electricity, but they can also be found on a smaller scale as isolated pockets in many other domains. Kuhn used the terms paradigm and normal science to express his concept of what makes these domains different from, say, astrology. Kuhn’s contribution is described in the following section and its relevance to SPE+ is explained in section 2.4.

2.3.3 Kuhn’s theory of ‘normal science’

Kuhn’s approach [46] to the question of how scientific knowledge develops is helpful for understanding evolution in software systems. His primary concern was to explain a pattern of knowledge development that seemed characteristic of sciences such as chemistry and the various disciplines within logic and mathematics. This pattern consists of successive periods of what Kuhn called normal science that each take place within a particular framework or paradigm. A paradigm, in this context, is

‘the general theoretical assumptions and laws and the techniques for their application that the members of a particular scientific community adopt’ [47, p. 108].

This use of the term paradigm is more specific than its everyday meaning, and in Kuhn’s theory it became an elaborate concept with several related senses.

Kuhn’s core concept of paradigm was that during conditions of ‘normal science’, a discipline has a single, accepted body of knowledge that is taught to practitioners, defines their research programme, and guides their methodology. Analogously, from a requirements analysis perspective, a paradigm also defines the conceptual framework or theory that must be modelled by, and can be taken for granted by, software systems that model knowledge in the domain of that paradigm.

Kuhn also identified that the process of ‘normal science’ within a discipline is occasionally disrupted by episodes of crisis that fundamentally change the way in which the discipline is defined and practised. Sometimes the crisis takes the form of a ‘scientific
revolution’ in which the previously accepted paradigm is overthrown and replaced by a new one. Kuhn’s examples include the demise of alchemy and its replacement by modern chemistry. Some crises have a more limited scope, resulting in the partial replacement of a paradigm. When such upheavals occur in the processes of ‘normal science’, corresponding changes must be made in software systems that depend on the affected paradigm.

Characteristics of paradigms

A major contribution of this work is to argue that there is a correspondence between Kuhn’s concept of paradigm and the $P$ category in SPE+. This section describes the relevant aspects of Kuhn’s work and draws attention to Masterman’s important contribution to clarifying Kuhn’s concepts. The relationship with SPE+ is explained later, in section 3.2.2.

When Kuhn was developing his theory of ‘normal science’, he needed to name his innovative concept that describes the distinctive, unifying character of a scientific discipline. He chose to use the term paradigm and to extend its everyday meaning. This has sometimes led to misunderstandings about the nature of Kuhn’s concept, and subsequently Kuhn regretted [48] that he had not chosen a more distinctive name for it. However, by then the extended meaning of paradigm had become widespread and it has continued to be used by both scientists and philosophers of science.

Masterman [49] identified three distinct but related senses or aspects of paradigm in Kuhn’s work. They are summarised here from Masterman’s descriptions, and illustrated with examples from object-oriented software engineering.

Construct aspect — a technique, instrument, language or model that is used to solve puzzles. This is the most concrete sense of paradigm in Kuhn’s work and the closest to the everyday meaning of the term.

Example: the Unified Modeling Language (UML) enables software designers to ‘specify, visualize, and document models of software systems, including their structure and design’ in a standardised way using object-oriented concepts.

Sociological aspect — the processes and organisations that are used by the experts in a discipline to sustain a consensus on its guiding principles and to systematically teach them to new entrants. Masterman identified the sociological sense of paradigm as Kuhn’s most innovative contribution to understanding the scientific process.

Example: the object-oriented software community organises journals and conferences, e.g. ECOOP, OOPSLA, that perform important sociological functions. For example, the peer review process tends to encourage innovation provided that it appears constructive within the existing paradigm. It protects the community’s core values, knowledge and achievements by expecting scholarly contributions to acknowledge and build on relevant prior work.

Metaphysical aspect — a partial world view or theory that identifies a discipline and provides it with a distinctive conceptual framework. The metaphysical sense of paradigm includes not only the explicit, formalised, theory of a discipline but also its assumptions, which often seem so obvious, at least to its practitioners, that they are taken for granted.

Example: Fowler [50] discusses some of the differences between the assumptions that are implicit in object-oriented design notations and those of other methodologies that are needed to understand business entities and relationships.

Paradigm formation processes

Masterman observed that, historically, scientific paradigms usually originate from the Construct sense, described above. Someone discovers a way of solving a problem that was previously intractable, or not even recognised as a problem. The discovery doesn’t ‘fit’ into existing paradigms but it opens up new opportunities for doing ‘normal science’ and consequently wins adherents. For example, the Copernican revolution in cosmology can be traced to two kinds of constructs. First, Copernicus and Kepler discovered that astronomical calculations could be simplified by assuming, respectively, a sun-centred universe and elliptical planetary orbits. Both assumptions contradicted the existing paradigm for cosmology and, initially, could only be justified on the grounds that they worked. Second, the invention of the telescope enabled Galileo to make observations, e.g. of Jupiter’s satellites, that also did not fit in the existing paradigm.

When a new paradigm emerges in this way, it initially justifies itself by its practical success in solving puzzles, particularly those that the previous paradigm could not solve or did not recognise. In the early stages of a paradigm’s development, its theoretical basis is often weak or even non-existent — it works as a technique but its advocates cannot adequately explain why this is so. Over time, the processes of ‘normal science’ may elaborate the paradigm by developing it in the metaphysical sense, and institutionalise the paradigm by creating organisations that sustain it in the sociological sense. In the case of the Copernican revolution in cosmology, Newton’s laws of motion provided its metaphysical basis until they were superseded by the concepts of Einstein’s theory of general relativity. An example of the sociological sense of paradigm is the foundation of the Royal Observatory, Greenwich by Charles II in 1675.

One of the effects of these processes is that paradigms tend to develop a stable, hierarchical structure. The paradigm’s practitioners gradually identify the ideas that provide the foundations of their discipline, and usually they try to express these ideas in the form of laws and principles. Lakatos [51] examined this aspect of the natural sciences. He found hierarchical structures of knowledge that were broadly divisible into a ‘hard core’ of fundamental theories and principles, and a ‘protective belt’ of progressively less certain and revisable knowledge and observations that depends on the hard core. The processes of ‘normal science’ produce continual incremental changes within the protective belt of a paradigm. Simultaneously, the sociological role of the paradigm prevents fundamental change within the hard core unless the case for it is overwhelming, resulting in a Kuhnian ‘scientific revolution’. Similarly, Kuhn observed that the processes of ‘normal science’ in disciplines with mature, successful paradigms tend to constrain the evolution of the paradigm to incremental changes that fill in gaps rather than overturn previous results.

The processes of paradigm formation are not inevitable and need not proceed uniformly in every knowledge domain. In some domains, one or more of the three senses of paradigm remain unfulfilled or there are competing candidates. In this work, such domains are said to be non-paradigmatic in terms of Kuhn’s and Masterman’s criteria. For example, the domain of astrology has neither a governing body that sets standards and trains new entrants, nor an overarching consensus on what constitutes ‘good practice’.

Masterman also described a transitional category between non-paradigms and paradigms, referred to in this work as emerging paradigms. During the process of paradigm formation, a domain may present competing alternatives for its techniques, metaphysical assumptions and social organisation. This diversity may lead to the development of distinctive, and often competing, ‘schools’ within a domain, e.g. the various approaches to psychotherapy.

In this work, the term pre-paradigmatic will be used to refer collectively to non-paradigmatic and emerging paradigm domains. In section 3.2.1 it will be argued that there is a correspondence between pre-paradigmatic domains and the $E$ category in SPE+. 
2.4 Requirements Analysis, Paradigms and Hermeneutics

We now consider how the concepts of hermeneutics and paradigm apply to the requirements analysis process. General considerations that apply to all real-world software, i.e. E-type and P-type systems, are described below. There are also significant differences, depending on whether the application domain is paradigmatic or pre-paradigmatic. These issues are discussed in two subsections.

Ideas that inform stakeholders’ requirements for a software system can be drawn from various sources. For example, they can be derived from paradigms, in Kuhn’s extended sense of that term, and from various kinds of pre-paradigmatic knowledge, including ‘common sense’ and ad hoc notions. In practice, stakeholders often express their goals, assumptions and requirements in the form of scenarios and exemplars. Consequently, an analyst must use a discovery process to find the underlying theories and assumptions. An analyst’s aim should be to achieve a sufficient understanding of the viewpoints that different stakeholders have implicitly adopted. In other words, the theories that stakeholders hold about the real-world domain of a system have to be inferred from the partial information that is available at the time that the analysis is carried out.

The interpretation of stakeholders’ theories involves an unbounded set of assumptions that are made by both analysts and stakeholders. Assumptions about software systems arise from the abstraction, reification and bounding processes that are essential to reduce the unbounded number of properties of any real world domain to a bounded set of requirements that can be implemented in a system.

Some assumptions may be known from the start of the requirements analysis process, and others may be discovered during it. Using requirements engineering techniques\(^{10}\) should improve the discovery rate but an unbounded number of implicit, unrecognised assumptions will always remain. The subset of assumptions that stakeholders are aware of will tend to change continually over the lifetime of a system. Of course, many assumptions are, at least initially, irrelevant to the development and use of the system. However, as the world changes, some previously irrelevant assumptions may become relevant and some that were previously valid may become invalid. All assumptions are potentially a source of unexpected program behaviour and unacceptable or incorrect results. Many failures of IT systems and projects have been attributed to assumptions that either became, or were always, incorrect.

2.4.1 Requirements analysis in paradigmatic domains

In paradigmatic domains, an analyst can validly use methodological hermeneutics. In such domains, requirements analysis must consider both the resources provided by the paradigm, and the statements made by stakeholders from their various perspectives. Depending on a paradigm’s stage of development, it may provide constructs, theories, assumptions, organisations, etc. that an analyst can mine for domain knowledge that is relevant to a software system. This knowledge can be used by an analyst in several ways:

- to derive a baseline model of the domain that a system’s stakeholders may wish to extend
- to validate stakeholders’ descriptions of the domain
- to identify stakeholders’ theories that are idiosyncratic with respect to the accepted paradigm for the domain

That is to say, the requirements analysis process within a paradigm can and should use the particular methodological hermeneutics defined by that paradigm.

\(^{10}\) Lamsweerde [28] provides an interesting summary from the viewpoint of goal analysis.
This approach is most effective in the kinds of scientific domain that Kuhn studied but it is not restricted to the natural sciences. It can be applied validly in other domains that share Kuhn’s descriptions of the ‘normal science’ mode of inquiry and conform to Masterman’s three roles of paradigms. For example, Wernick [52] and Wernick and Hall [53] examine whether the software engineering discipline is paradigmatic in Kuhn’s terms.

However, outside the natural sciences, paradigms that are well-formed in the Kuhn-Masterman sense usually have a very restricted scope and are difficult to compose into the larger bodies of knowledge and theory that characterise sciences such as chemistry and biology. Non-scientific disciplines are more likely to have one or more paradigmatic ‘islands’. For example, natural languages use many different alphabets and writing systems. They have evolved through complex cultural, rather than scientific, processes. Nevertheless, the Unicode standard attempts to solve a specific problem — encoding different natural languages in software — by creating a framework that mimics many features of a scientific paradigm. The Unicode ‘paradigm’ includes problem-solving constructs, an organisation that manages the Unicode standard, and a set of concepts for understanding its domain. It is, however, an isolated paradigm-in-the-small compared to, say, the role of the Periodic Table in chemistry.

The scope of a particular software system need not coincide with a single paradigm. Consequently, in practice, the requirements of many scientific and most non-scientific software systems refer to multiple paradigms and also to pre-paradigmatic knowledge. Nevertheless, analysts can consider using the paradigm concept to modularise requirements and should be aware that different hermeneutics are appropriate within a Kuhnian paradigm and in the absence of such paradigms.

2.4.2 Requirements analysis in pre-paradigmatic domains

In pre-paradigmatic domains, requirements analysis must rely primarily on phenomenological hermeneutics. In such domains, the kinds of resource that can be used within a methodological hermeneutics are less extensive, or less reliable as sources of domain knowledge, or they may not exist. In pre-paradigmatic domains, the process of discovering the relationship between the real world, the description of a problem, and a software model of the problem must rely primarily, and sometimes wholly, on skillful interpretation of stakeholders’ statements. That is to say, an analyst must use phenomenological hermeneutics and dialogue to discover what stakeholders ‘really’ mean.

Similarly, stakeholders of systems in pre-paradigmatic domains must also rely on these processes if they wish to assess whether an analyst has reached a sufficient understanding of the domain to produce an adequate software model of it. The phenomenological aspect of this process is that the participants should ‘adopt a stance of critical self-understanding’ [42] that recognises that everyone brings their own subjectivity, assumptions and concerns to the dialogue.

Mallery et al. [42] draw attention to Ricoeur’s [54] distinction between discourse and dialogue which is relevant in this context. For Ricoeur, discourse is a more detached, impersonal process that occurs when an interpreter engages with a text, e.g. an analyst tries to understand a published standard. Dialogue is more interactive and more clearly related to a specific situation. In the context of requirements analysis, dialogue includes the possibility of negotiation between the speaker and the interpreter, and also allows for case-based justifications.
3 SPE+

3.1 Introduction

This section describes the proposed refinement of the SPE categories and shows how they can be defined in terms of the key concepts that were introduced in section 2, namely replicator, hermeneutics and paradigm.

SPE+ includes several significant innovations over earlier presentations of SPE:

- SPE+ asserts explicitly that the $E$ category represents the default case for the evolution of software systems.
- SPE+ defines the $P$ and $S$ categories as special cases that arise from certain kinds of stakeholders’ requirements.
- SPE+ replaces the ambiguous definition of the $P$ category in SPE with a definition of $P$-type systems that is derived from Kuhn’s concept of ‘normal science’ and the Kuhn-Masterman concept of paradigm. To recognise this change, ‘P’ stands for Paradigm-based in SPE+.
- The definitions and descriptions of the categories in SPE+ are derived not only from the domain of software engineering but also from relevant philosophical traditions and theories of generalised evolution.

Nevertheless, SPE+ retains the spirit of the original definitions of the $S$, $P$ and $E$ categories and does not conflict with the earlier work. The effect of the SPE+ refinements is to make some of the implicit aspects of the original descriptions explicit and more specific.

3.2 The SPE+ Taxonomy

3.2.1 E category – ‘Evolving’

Defining characteristics of $E$-type systems

In SPE+ the default case of evolution in software systems is represented by the $E$ category. Unless there are exceptional circumstances, which are described later under the $P$ and $S$ categories, a software system will tend to evolve continually during its productive lifetime. Conversely, $E$-type systems that do not evolve for some exceptional reason, e.g. resource shortages, inflexible architecture, will tend to become progressively less useful. Lehman summarised this relationship in ‘Law I of Software Evolution – Continuing Change’ [22]. The tendency for $E$-type systems to continually evolve has implications for a system’s stakeholders, its architecture and its global software process.

The distinguishing characteristics of the SPE+ categories can also be described from perspectives that are not centred on engineering concerns. So in terms of Kuhn’s paradigm concept (§2.3.3), the distinguishing feature of the $E$ category is that each system’s requirements are wholly or partially drawn from one or more pre-paradigmatic domains. That is to say, if a set of requirements has dependencies on knowledge from domains that are not paradigmatic in the sense described by Kuhn and Masterman (§2.3.3), then any system that implements those requirements will be $E$-type.

Section 2.4.2 identified an equivalence between pre-paradigmatic domains and the use of phenomenological hermeneutics to analyse stakeholders’ requirements. It follows that the process of developing a conceptual model for an $E$-type system involves making judgements within an iterative, interpretative process. This process necessarily uses phenomenological hermeneutics and dialogue, both between stakeholders to resolve any conflicting requirements, and between stakeholders and the analyst to reach a shared understanding.
The discipline of software engineering provides an analyst with general guidance and techniques, e.g. Parnas’s information hiding principles [55], but they can only be used effectively in conjunction with detailed knowledge of the domain. That is to say, mathematical and engineering techniques can help an analyst to work systematically but they cannot be substituted for hermeneutic interpretation of domain knowledge that must usually be gained through dialogue with domain experts and other stakeholders. Some of the analyst’s tools and methods may be paradigmatic within software engineering [52, 53], but their application to a pre-paradigmatic domain cannot completely eliminate choice from the analysis process or uncertainty from its conclusions. This implies that there are limits on the application of formal methods to \textit{E-type} systems.

\textbf{E-type systems and the behaviour of replicators}

The SPE+ categories can also be understood in terms of the concepts of replicators and vehicles (§2.1.1). Every evolving software system can be seen as a vehicle for a collection of replicators that have \textit{differential} survival rates. That is to say, the replicators within an evolving system vary in their success rates for getting copied into the next release or into another system or artifact, and for avoiding being discarded. However, each replicator does not necessarily behave independently in this situation. The survival chances of a replicator often depend partially on the survival of its ‘neighbours’ because, in practice, replicators in software tend to be copied or discarded in related groups. An analogy is the phenomenon of ‘gene linkage’ [5] that is found in living things.

Many of the linkages between replicators found in software follow from the requirements of the system. For example, in a learning management system, a concept of \textit{assessment credit} may depend on a concept of \textit{course}. Thus the survival chances of the ‘assessment-credit-concept’ replicator have become linked with those of the ‘course-concept’ replicator. If the ‘assessment-credit-concept’ replicator gets copied somewhere, it is very likely that the ‘course-concept’ replicator will travel with it. Linkages can also arise because a particular combination of replicators does something that is useful in a domain-independent way, e.g. the various useful combinations of design patterns that have been identified by Gamma \textit{et al.} [32]. Some of these combinations, e.g. patterns relevant to software frameworks [56], have been proposed as ‘pattern languages’ [13], increasing the chances that their constituent replicators will get copied as a group.

The distribution of replicator linkages, in terms of their number, size, strength and other properties, will vary between systems. At one extreme, all the replicators in a system can be tightly bound in a single linkage that is copied or discarded as a unit; it will be seen later that this is characteristic of \textit{S-type} systems. At the opposite extreme, each replicator’s chance of survival is independent of every other replicator.

The distributions of replicator linkages that can be found in \textit{E-type} systems are likely to be very diverse. The dominant characteristics of \textit{E-type} systems — multiple pre-paradigmatic domains, unstable environment, feedback from system use to evolving requirements — tend to increase divergence in the survival rates of replicators. Conversely, in the absence of the control exerted by a paradigmatic domain, it is very unlikely that all or most of the replicators will be strongly linked together as far as their survival chances are concerned. Thus, it is more likely in an \textit{E-type} system that there will be a large number of small groups of linked replicators and that many of the linkages will be relatively weak and transient. The linkages are more likely to depend on the requirements and design choices of each system, and less likely to depend on paradigms and externally defined standards. Thus the evolution of an \textit{E-type} system, considered in terms of the outcome of the differential survival of the replicators that it hosts, is likely to have a high number of degrees of freedom. This is because there will usually be many possibilities for the survival rates of its replicators to differ from each other.
The replicator-based perspective described above is consistent with the Laws of Software Evolution (Table 1 in section 2.1.3). An $E$-type system must be continually adapted to maintain stakeholder satisfaction (Laws I, VI). Each adaptation provides opportunities for ‘indigenous’ replicators to get copied into the next release, and for ‘migrant’ replicators to enter the system. It also presents threats of ejection to indigenous replicators, as a result of refactoring or rationalisation (Law II). However, the turnover in a system’s population of replicators is constrained (Laws III, V), and the capacity of a system to absorb new replicators is also limited (Laws II, IV). Therefore, there will be competition between replicators for survival and hence differential survival rates.

**Uncertainty in $E$-type systems**

The characteristics of pre-paradigmatic domains, and the unavoidable use of phenomenological hermeneutics to analyse the requirements of $E$-type systems, contribute to uncertainty in the global software process of $E$-type systems. This is part of Lehman’s ‘Software Uncertainty Principle’:

> ‘In the real world, the outcome of software system operation is inherently uncertain with the precise area of uncertainty also not knowable.’ [57]

Lehman [57] identified three primary sources of uncertainty in the results of programs; one of these sources, ‘Pragmatic’ uncertainty, arises because $E$-type systems are

> ‘finite models of an unbounded, effectively infinite universe of discourse, and one can never be certain that one has identified all necessary assumptions.’ [57]

This is equivalent to saying that to understand the relationship between an $E$-type system and its domain requires phenomenological hermeneutics.

**Implications for global software processes**

From the viewpoints of its stakeholders, an $E$-type system often appears to be in or to approach a state of continual change (Law I of software evolution). An $E$-type system’s evolution is affected by both organisational and engineering processes, which sometimes appear to act independently, sometimes harmoniously, and sometimes opposing each other. These processes are themselves affected by the interpretations, decisions and actions that stakeholders make from time to time. For example:

- stakeholders in $E$-type systems can define and redefine problems without referring to the constraints of an accepted paradigm;
- stakeholders’ requirements for the scope of a system are more open to reinterpretation and revision when there is no accepted paradigm to provide reference cases.

The global software process is also affected by the environment in which $E$-type systems operate, which changes continually in both predictable and unexpected ways.

Consequently, the global software process of an $E$-type system usually exhibits several levels of feedback, for example:

- When a system is brought into use, stakeholders may experience feedback about inconsistencies between their theory of the problem and its models in both the system and the real world. They may also notice unexpected discrepancies between the software model and the real world.
- The assumptions about and approximations of the real world in the software model may have become less acceptable to stakeholders since the time when the system was planned because the real world has been changed in several ways by the system development process itself.
– the real world now includes a new or revised software system;
– the domains in which the system operates have changed as a consequence of designing, building, installing and operating the system;
– any change in the use of the system tends to produce side-effects in the interactions between stakeholders and other real-world entities outside the software system.

For example, if a form in a business process is made available in an electronic format, this may reduce clerical errors because the electronic form can actively check for some kinds of error. However, it may also raise issues about other aspects of the business process, such as automated routing of forms and paperless authorisation procedures.

– non-functional aspects of the system may interact with entities and processes in the system’s environment.

For example, stakeholders may require the security aspect of a system to conform to externally defined standards and processes which evolve according to their own dynamics. Thus many e-commerce systems have required adaptations, not only to fix their own security loopholes but also to take account of external changes, e.g. in the online payment systems that they collaborate with, in the practices required by their bankers and credit agencies, and in regulatory frameworks. Many of these changes are themselves responses to the spread of online shopping and its side-effects, such as new opportunities for fraud.

In pre-paradigmatic domains, the stakeholders of a software system can respond to feedback by adjusting their theory, and subsequently the system, in ad hoc ways to fit their revised perceptions. Conversely, in paradigmatic domains the processes of ‘normal science’ create strong pressures to protect core theories and to resist adjustments that merely accommodate anomalous observations (see §2.3.3).

Architectural implications

The uncertainty that is associated with E-type systems has architectural implications. In general, E-type systems must be expected to evolve, and it therefore becomes very important for stakeholders to consider whether a system is likely to be adaptable to changing circumstances. In many cases, the architecture of an E-type system will be an important factor in maintaining stakeholders’ satisfaction with the system. This is because one of the roles played by a system’s architecture is to define which system properties are adaptable and which are fixed (see §2.2.2).

For any particular adaptation to a system, the properties to be changed and preserved may be specified at various levels of architectural abstraction, and the ownership of the specifications may cut across organisational boundaries. An example of a Contextual (see Table 2 in section 2.2.2) level requirement is the identification and protection of the critical success factors for a software system, e.g. that the system conforms to the system conforms to a particular interface. In the genealogical domain, for example, software products are often judged on the quality of their support for the GEDCOM11 de facto standard for data interchange. The specification of this interface evolves with its own dynamics, which are influenced by many stakeholders, who include both suppliers and users of genealogical software products. The interface is specified at the Logical and Physical levels using BNF syntax. However, at the Conceptual level, the meaning of the various data types is described informally. At the Component level, the relevant program code in a conforming product might be distributed over several modules. Thus, a stakeholder in an E-type software system often

---

11GEDCOM Standard 5.5 has been published at various, often ephemeral, websites including http://homepages.rootsweb.com/~pmcbride/gedcom/55gctoc.htm
needs to track the co-evolution of several specifications and objectives that are expressed in different languages and at different levels of abstraction.

It might appear that ideal E-type systems would be designed with ‘separation of concerns’ [58, 55] at multiple levels of architectural abstraction, and with explicit links between related requirements at different levels of abstraction. However, in practice, system designers and architects will usually find that tradeoffs and compromises are unavoidable. Some stakeholder concerns, notably security and performance, tend to cut across all other concerns. Highly elaborate separation of concerns may make a system difficult to understand. The technique of aspect-oriented design has been proposed [59] to mitigate these problems. Nevertheless, E-type systems always contain the possibility that an assumption, either consciously made or unexamined, will become invalid. Theories of software evolution imply that there are no perfect solutions to these issues.

3.2.2 P and S categories: common features

The P and S categories of software systems are special cases where stakeholders have made explicit policy decisions that affect the kinds of evolution that can occur in the system. The effect is to reduce, or even remove, the influence of some sources of evolution that are found in E-type systems. The particular decisions that lead to the creation and perpetuation of P- and S-type systems are explained in sections 3.2.3 and 3.2.4 respectively.

The decisions that define P- and S-type systems have to be made and enforced explicitly by stakeholders. If the decisions are implicit or the conditions arise accidentally, then it is much less likely that they will be complied with consistently over a system’s lifetime. They must also be policy decisions. In terms of Zachman’s taxonomy (Table 2 in section 2.2.2), the stakeholders of P- and S-type systems have identified a strategic requirement, i.e. at Zachman’s ‘Contextual’ level, to restrict the possible evolution of a system. Otherwise, it is likely that the decision will be neglected, or traded-off against other concerns, or made irrelevant by events, which would produce an E-type system.

In practice, the strategic decisions that characterise P- and S-type systems also have to be embedded in both a system’s architecture and its global software process. Otherwise the system is likely to become progressively more like an archetypal E-type system. Some of the implications are considered in more detail below.

3.2.3 P category – ‘Paradigm-based’

Defining characteristics of P-type systems

The previous section distinguished P- and S-type systems from E-type systems by their association with strategic decisions that restrict the possible evolution of a system. The additional property that distinguishes a P-type system from the S category is that the satisfaction of its stakeholders depends on the system maintaining consistency with a single paradigm over the system’s lifetime. An example of a large-scale system that is based on a single paradigm is the Virgo [60] simulation of the evolution of the universe. In this case the relevant paradigm is the laws of physics as they apply on a cosmological scale. The success of Virgo depends both on the accuracy of its results, compared to astronomers’ observations, and on its consistency with gravitational theory.

This kind of dependency can also be made on an external standard that is treated as a paradigm by the system’s stakeholders. For example, during the 1990s British Telecommunications plc operated a software system [61] in its telephone exchanges that provided an interface between analogue subscriber line switches, which conformed to a written specification, and its digital network, which implemented an international standard. The success of the system depended on its conformance to these specifications, which played the role of a paradigm in this context.
Thus the evolution of a $P$-type system is constrained by the strategic decision of its stakeholders to keep the system consistent with a paradigm. This constraint will be experienced in two ways. It will prevent some kinds of change that might otherwise have occurred. It may also induce change, either when the paradigm is updated or when opportunities arise, e.g. through technological change, to improve the system’s consistency with its paradigm.

Stakeholders of systems in paradigmatic domains are more constrained than stakeholders of $E$-type systems in changing their theory of any specific problem within that domain. A paradigmatic domain provides an overarching conceptual framework that has a significant degree of internal coherence and discipline and is shared with other experts in that field of knowledge. Consequently, the task of understanding how to model a particular problem can use the methodological hermeneutics that the paradigm defines. The discipline’s paradigm defines both a general model for the problems that are within its scope, and techniques for applying the generic model to a specific case. The paradigm also restrains stakeholders from making piecemeal or arbitrary adjustments to either their ‘local’ theory or its software model, if this would reduce the credibility of the system within the paradigm’s community. The importance of this distinction between $E$-type and $P$-type systems is not undermined by the fact that paradigms themselves evolve in limited ways (see §2.3.3).

Thus for a $P$-type system, the domain’s paradigm predefines a complex structure of assumptions, theories and techniques that are already familiar to many of the system’s stakeholders. This has effects on the feedback loops within the global software process, particularly from the use of a software system back to its requirements [29]. In the case of $P$-type systems, this loop is indirect and mediated through the processes of scientific discovery, peer review etc. That is to say, using a $P$-type system can only change the paradigm that defines the system’s concepts if use of the system leads to the acceptance of new scientific knowledge into the same paradigm. The closer the concept lies to the paradigm’s ‘hard core’, the less likely it is that the paradigm’s community will decide to change it.

**Replicator behaviour in $P$-type systems**

The dependence of $P$-type systems on paradigmatic domains can also be expressed in terms of replicator behaviour. It produces a situation where the domain-specific replicators in a software system have their survival chances yoked together by depending on a paradigmatic domain. Furthermore, replicators that are domain-independent will also find that their survival chances are influenced by the extent to which they help the system to conform to the paradigm. Thus the differential survival of replicators in a $P$-type system tends to be dominated by a single persistent source of evolutionary pressure, whereas an $E$-type system would usually have multiple, possibly competing, evolutionary pressures that might change over time.

**$P$-type systems and software reuse**

The characteristics of the $P$ category give it a strong association with software reusability. Many design techniques for promoting software reuse can be understood in SPE+ terms as attempting to disentangle coherent $P$-type components from each other and from $E$-type noise. The pattern catalogues in, for example, [11, 32] provide repertoires of reusable $P$-type solutions to software design problems. They are largely independent of specific implementation languages and application domains. Fowler’s use of analysis patterns [50] implicitly abstracts $P$-type information elements from the complexity and incidental detail of real-world, $E$-type, domains. Edwards et al. [62, 8] proposed a similar approach to the design of ‘immortal’ software.
Simon’s work [63] on evolution in both natural and artificial systems implies that reusing components should tend to reduce the costs of system evolution. P-type components at any level of architectural abstraction can play the role of Simon’s stable intermediate forms. Simon used this term to refer to subsystems that can be used as building blocks or components, i.e. they can be used and reused in the evolution of more complex, but possibly less stable, systems, including E-type systems.

Many systems constructed from P-type components will also have the quality that Simon called nearly decomposable [63], i.e. interactions between the system’s subsystems are weak but not necessarily negligible. Consequently, nearly decomposable systems tend to have more predictable behaviour at both the subsystem and aggregate levels, and over different timescales. This can be contrasted with E-type systems, where it is common to find that changing a subsystem tends to cause ripples of consequent changes.

### 3.2.4 S category – ‘Specification-based’

**Defining characteristics of S-type systems**

The E and P categories in SPE+ define two idealised types of evolving systems. Real-world software systems can be expected to conform to these types to a greater or lesser extent. The S category is somewhat different. As explained in section 2.1.4, the S category defines the conditions in which software evolution does not occur. These conditions are very restrictive and, in practice, few fully conforming S-type systems are found.

The condition that is necessary to prevent the occurrence of software evolution is that the sole criterion of stakeholder satisfaction with a software system is its correctness with respect to a specification. If stakeholders care about any property of the system that has not been completely specified, then it is likely that the system will evolve for the reasons explained in previous sections, and the system will therefore be P- or E-type. In particular, if stakeholders care about a system’s relevance to the real world, then the system will not be S-type.

The centrality of specifications to the S category implies that S-type systems are more likely to be based on paradigmatic domains, particularly mathematics and logic. Nevertheless, in principle, the specification of an S-type system can be drawn from any or no domain. The essential property of an S-type system is that, once the specification of its requirements has been decided, the specification must be divorced from any paradigm or theory that it was derived from and must be treated as axiomatic. That is to say, any future evolution in a ‘parent’ paradigm cannot be allowed to affect the specification, which must be self-sufficient and the only criterion for judging the system. To restate points made earlier in the context of E- and P-type systems, if a system’s specification retains dependencies on any paradigm, then the system cannot be S-type. If the dependency is a strategic decision by stakeholders, then the system can be P-type, otherwise it will be E-type, and its evolution will be subject to the appropriate dynamics.

**Architecture and design in S-type systems**

Further limitations on the practicality of S-type systems arise from the relationships between program size, complexity and design. In practice, as Dijkstra [64] observed, design tradeoffs cannot be ignored. As a program increases in size and/or complexity, it becomes more difficult to prove its correctness unless the designer has structured the program to facilitate the relevant kinds of proof. Hence, in practice, a stakeholder requirement for correctness also implies design choices that will permit correctness to be demonstrated.
Replicators in S-type systems

The replicators that are found in S-type systems include functions and algorithms, and also more abstract artifacts such as the design styles and patterns alluded to by Dijkstra. By definition, an S-type system does not evolve. Therefore, differential survival of replicators does not occur within an S-type system but it can occur within collections of formally specified systems, which may include S-type systems. Replicators can be copied from one formally specified system to another as part of the process of writing a new program, which could be S-type. When replicators are copied between formally specified systems, this produces the effect that Dijkstra called a 'system family' [64, pp. 39–41], in which a collection of similar programs share a common abstract specification but differ in their concrete specifications.

3.3 Validation of SPE+

As far as the authors are aware, the available evidence is consistent with SPE+ and illustrative examples have been mentioned where appropriate. However, this is insufficient to make a compelling argument and more extensive and rigorous tests are required to establish whether SPE+ is valid.

SPE+ should be used to infer hypotheses and models about possible differences in the observed evolution of E- and P-type systems. Section 3.2 makes many allusions to the impacts on stakeholders, global software processes, system architecture, etc. that can be expected when software systems evolve. We are already conducting further research in this area by investigating whether E-type and P-type modules can be identified within systems, and if so, whether they exhibit differences predicted by SPE+. For example, the current research of three of the authors (Cook, Harrison and Wernick) is focussed on a case study of an industrial-scale telecommunications system. It is desirable that similar tests should also be carried out under controlled, laboratory conditions where this is feasible.

The validity of SPE+ can also be considered in a broader sense, by considering its implications for various kinds of software processes, the How? of software evolution. The following paragraphs illustrate this by sketching some issues that could be explored using SPE+.

SPE+ and stakeholder policy decisions

Lehman’s initial exposition of SPE emphasised the importance of user satisfaction in the dynamics of software evolution. SPE+ reinforces this by identifying stakeholders’ strategic decisions as the critical factor in creating and perpetuating P- and S-type systems. This should encourage further research into the global software process and its relationship to policies and practices in IT governance.

SPE+ and open source software

Section 2.1.1 showed that the concept of replicator can be used to identify theoretical differences in the evolutionary potential of open-source and proprietary software products. Previous empirical research, e.g. [2], has suggested that open-source software processes may exhibit variations from the classic form of Lehman’s Laws. These lines of research have yet to be coordinated with SPE+. So it remains an open question whether an observed difference in evolutionary behaviour between software systems should be attributed to:

- the open/closed character of the software process, or
- the SPE+ category of the product, or
- a combined effect of the categories of the process and the product, or
• other factors.

Such research would help to establish the relative importance of the SPE+ category in determining the likely evolution of a software system.

SPE+, design patterns and software reuse

An implication of SPE+ is that $E$- and $P$-type systems have different architectural properties and thus may be suited to different kinds of design patterns. $P$-type components appear to be good candidates to fulfill the role in evolving systems that Simon called \textit{stable intermediate forms} [63] (§3.2.3). Conversely, designing $E$-type components to have lower evolution costs will usually be more challenging. Their association with pre-paradigmatic domains increases the probability of changes to their architectural properties at the ‘Conceptual’ and ‘Contextual’ levels, which makes them less suitable candidates for stable intermediate forms. Andrade et al. [65] have suggested that design patterns based on concepts of coordination and superposition may be effective for separating volatile business rules from ‘immortal’ [62, 8] properties in $E$-type systems. Concepts of aspect-oriented design [66] are also relevant.

SPE+, agile methods and architectural concerns

Alternatively, there may be a case for designing some $E$-type components to be cheaply disposable, rather than modifiable. The higher rate of churn in the design of Web interfaces, compared to their underlying information sources, appears to illustrate this. However, this approach may increase the risk that architectural principles and other strategic stakeholder decisions will be neglected. One of the challenges facing the advocates of both ‘agile methods’ and ‘enterprise architecture’ is how to reconcile their conflicting concerns in conditions of great uncertainty and rapid change for software systems. SPE+ provides a set of concepts and definitions within which these ideas can be investigated.

4 Conclusions and Future Research

This work is primarily concerned with refining the metaphysical aspect of the emerging paradigm of software evolution, to use the terminology of section 2.3.3. We have described a set of concepts, drawn from software engineering and other domains, that provide the foundations for understanding what software evolution is. We have demonstrated the value of this conceptual framework by using it to propose refinements to the SPE taxonomy of evolving software systems.

The revised form of the SPE taxonomy, presented here as SPE+, addresses some perceived weaknesses and ambiguities in the original formulation. SPE+ provides a basis for classifying evolving software systems that demonstrates a unification of concepts drawn from software engineering, from generic theories of evolution, from the hermeneutic tradition in philosophy, and from Kuhn’s concepts of paradigm and ‘normal science’. The strongest aspect of the original SPE taxonomy, namely its insights into evolution in $E$-type systems, has been retained and is entirely consistent with the refinements of the $P$ and $S$ categories in SPE+.

The focus in this work is on theory-building. It needs to be complemented by more empirical approaches. In particular, it is important that further work should be done to develop testable models and hypotheses from the conceptual definitions and descriptions of the $E$ and $P$ categories. An important test of the validity and relevance of SPE+ will be whether it leads to different predictions for the evolution of $E$- and $P$-type systems and whether these differences are observable, both under laboratory conditions and in industrial-scale software systems. This work is currently in progress within our own
research groups and we hope that other teams will also feel encouraged to explore the implications of SPE+ and to test its validity.

The theoretical developments described in this work indicate that there is a continuing need for empirical studies of evolution in industrial-scale software systems. The diversity of software systems and software processes means that an extensive corpus of studies is required both to test conjectures and to suggest further refinements to theories. For example, the FEAST projects carried out pioneering work in this area but a limitation of those studies was that they concentrated on software development processes that were derived essentially from Royce’s waterfall [67] methodology. The theory of software evolution, including SPE+, predicts that its effects will also be found in software systems that have been developed using other approaches, e.g. iterative/incremental [68], open source [69], agile [70]. Valuable work has begun in these areas, e.g. Godfrey’s study [2] of evolution in open source software, but much more is needed. An important research aim should be to establish more precisely which phenomena of software evolution are universal, and which vary according to parameters such as the development method, the application domain, etc.

Such studies can take various forms. For example, case studies of well-documented, long-lived systems are valuable because they can provide opportunities to observe and measure a wide range of properties of a system, its global software process and its relationship to its application domain. There is also a role for studies that emphasise breadth rather than depth. For example, a collection of software system histories could be analysed using techniques such as case-based reasoning [71]. In contrast, classical experimental designs, i.e. involving control groups and statistically reliable sample sizes, present formidable difficulties in this research area. Given the practical difficulties of conducting non-trivial experiments in software evolution, this work is likely to require collaborative efforts. Empirical research in software evolution could be an interesting application area for a ‘semantic grid’ approach.

The concept of software evolution is gradually becoming accepted. The work reported here contributes to that process by showing that the theory of software evolution is capable of further development. In particular, this work uses SPE+ to show how the theory of software evolution can be integrated with other aspects of software engineering and with wider philosophical concerns.

The theory of software evolution also has implications for the practice, management and planning [39] of software development and adaptation. In particular, as software becomes ubiquitous, it will become increasingly important to be aware of the various assumptions — which may be inconsistent, out-of-date or simply wrong — that have been incorporated into software, and hence into products, processes and services that human society relies on. That is to say, as software becomes pervasive in everyday life, so too will the effects of software evolution. Unless we improve our understanding of its underlying processes, we are likely to be surprised by its emergent effects.

By balancing detailed investigations with broader perspectives, researchers into software evolution can help colleagues and IT practitioners to understand what software evolution is, why it happens, how it can be planned and managed, and how systems can be designed with evolution in mind — in short, how the benefits of software evolution might be realised and how its risks can be mitigated.

**Glossary**

**ADL** architecture description language

**CASE** computer-assisted software engineering

**COTS** commercial off-the-shelf

**DBMS** database management system
FEAST Feedback, Evolution And Software Technology
http://www.cs.mdx.ac.uk/staffpages/mml/feast1/index.html

semantic grid: a generic, easy-to-use infrastructure for e-science
http://www.semanticgrid.org/

UML Unified Modeling Language

Unicode: the universal character encoding standard used for representation of text for computer processing, http://www.unicode.org/

Bibliography


Acknowledgements

The work of Stephen Cook and Rachel Harrison is supported by the UK Engineering and Physical Science Research Council (grant no. GR/N01859) and the University of Reading.

The work of Paul Wernick is supported by the University of Hertfordshire.

32
Earlier work by Lehman and his colleagues was variously supported by the European Office of the US Army and by the UK Engineering and Physical Science Research Council.

The authors gratefully acknowledge the many helpful and incisive comments on draft versions of this work that they have received from colleagues and anonymous reviewers.

Authors’ Biographies

**Stephen Cook** is studying for a PhD degree in Computer Science at the University of Reading, where he is a member of the Applied Software Engineering Research Group in the School of Systems Engineering. His research interests are centered around software evolution. Prior to joining the University of Reading, he worked in the Information Technology Department at the Rutherford Appleton Laboratory and for Science Systems plc as an analyst/programmer. His information systems experience covers a wide range of research, administrative, and commercial projects. He holds a MSc degree in Computing (University of Wales) and a BA degree in Social Science (Middlesex Polytechnic).

**Prof. Rachel Harrison** is a Professor of Computer Science and Head of the Applied Software Engineering Research Group in the School of Systems Engineering at the University of Reading. Her research interests are centred around empirical software engineering and include software metrics, evaluating multimedia systems and applications, process modelling, requirements engineering, large-scale open systems, and software testing methodologies. Professor Harrison holds PhD and MSc degrees in Computer Science (from the University of Southampton and University College London, respectively), and MA and BA degrees in Mathematics from the University of Oxford. She is a Member of the British Computer Society and a Chartered Engineer.

**Prof. Meir (Manny) Lehman** is a Professor in the School of Computing, Middlesex University, where he continues his research, concentrating on developing a theory of computing systems and software evolution. In 1964 he joined the IBM Research Division, Yorktown Heights, where his projects included a study of the IBM programming process. This led to his discovery of the software evolution phenomenon. From 1972 to 2002 he was a Professor at the Imperial College of Science, Technology and Medicine, and Head of its Department of Computing for a five year term. He continued his research in the field of software evolution and led the FEAST/1 and FEAST/2 projects. He holds Batchelor and PhD degrees in Mathematics from Imperial College and a DSc degree from London University. He has published over 120 papers in the computing science and software engineering fields, holds some seven patents and lectures extensively the world over. He is a Fellow of the Royal Academy of Engineering (FREng) and a Member of the ACM, BCS, IEE and IEEE.

**Dr Paul Wernick** is a Senior Lecturer in Computer Science at the University of Hertfordshire. After qualifying as a Chartered Accountant, he gained experience of the development and use of commercial software as successively computer audit specialist, analyst/programmer and user support consultant. He holds a PhD degree from the University of London. His doctoral thesis examined from a Kuhnian perspective the belief systems underlying the theory and practice of software development. More recently, he has developed simulation models of a number of long-term software evolution processes. Dr Wernick’s current research interests are software evolution processes and their simulation, and the philosophical foundations of information systems and their development processes.