Software evolution in the age of component-based software engineering

M.M. Lehman and J.F. Ramil

Abstract: The eight laws of software evolution provide phenomenological descriptors of the evolutionary behaviour observed over the years in a number of software systems. The laws relate to E-type software, that is, software used to solve a problem or address an application in a real-world domain. Any computer application must undergo continuing evolution. The fact that the system implementing the application includes a significant number of components does not affect this fundamental truth. Thus one must expect the laws to be relevant in the context of component-based software engineering, though their statement, the phenomenology they reflect, may have to be refined. The most immediate implications of these observations are examined, their potential managerial impact discussed and some recommendations provided, thus giving a basis for questions and hypotheses to be further investigated and developed.

1 The move to CBSE, COTS and reuse

Component-based software engineering (CBSE) [1, 2], including the use of commercial off-the-shelf software (COTS) [3–5] is widely seen as a way of reducing some of the problems so often encountered, especially in ab initio software development. In particular, the use of COTS is seen as a means to achieve increased reliability and productivity and to decrease the delivery time for large-system implementation. Software development based on the combination of a set of existing components was discussed in the 1960s [6] but it was not until recently that the practical application has been widely explored in practice [1, 2]. A parallel is frequently invoked with other engineering fields, where the concept of components is almost universally accepted and applied. Nuts, bolts and a wide variety of electrical connectors and components exemplify internationally agreed standards. Such standard components can be obtained from numerous suppliers and used as building blocks in different and unique products.

Recent discussion of CBSE has focused on the how of component usage and integration [1, 2] and on principles and techniques for the construction of reusable components and component-based systems [4, 7]. As in the area of software evolution [8, 9], the what and the why of CBSE has received much less attention. Included in this omission are the attributes of CBSE products and processes. As shown elsewhere [8], the evolution of computer applications and of the software that implements them is inevitable. The latter may be achieved by changing a software to support the new functionality or performance requirements. Alternatively, these may be provided by throwing away the old system and providing a new system that has the desired characteristics [10]. In either event, understanding the way in which applications evolve is essential if effective and discipline evolution is to be achieved, and in particular, to facilitate planning, direction, management and control of CBSE technology and application.

A trend complementary to the use of components relates to software reuse, that is the use in some system of a software component originally developed for some other system or systems [7]. Application of reuse and, in particular, the increasing adoption of this technology, is considered by some [11] to be more effective in the long term, at least in some respects, than development of new code. However, extensive reliance on reuse introduces new problems. Some of these are similar to the problems that arise in CBSE and are implicitly covered by the discussion that follows. Others are beyond the scope of the present analysis.

The discussion in this paper reflects some of the findings of the FEAST/1 (1996–98) and FEAST/2 (1999–2001) studies. Their goals [12, 13] and a list of publications may be found at [14].

2 Laws of software evolution

The laws of software evolution [8, 14–20] encapsulate observed behaviour in a number of evolving systems studied over the years. In this sense, the laws provide a phenomenology, a description of systematically observed patterns. The term ‘laws’ was used to indicate that they emerge from sociological, organisational and cognitive phenomena which appear to be to a large extent beyond the control of individual software developers and even managers. The operative mechanisms are largely independent of the technologies being applied in the evolution process, hence the law designation.

The laws are not examined in detail in the present paper. It must, however, be noted that, after minor changes in detail and nuance of the relevant statements to take into account the understanding that has recently developed, at
least six of the eight laws are consistent with new data now available, with related process models and with interpretations of both [14, 20]. (Law IV is currently being re-examined; new data has been unavailable to assess law VII.) This greatly strengthens confidence in their overall validity for non-CBSE environments.

This paper examines the relevance of the laws in the context of component-based software in general and COTS-intensive systems in particular. The latter issue has also been raised by Hybertson et al. [3], who concluded that further empirical investigation is required. The analysis that follows contributes to the topic by raising questions that need to be considered in such an investigation and providing some provisional answers. It was originally developed [21] without knowledge of Hybertson’s et al. work and only addressed COTS-intensive processes. However, the conclusions are believed to be largely relevant to CBSE in general, and in particular when component suppliers and customers belong to independently managed organisations. (The terms customer and host organisation refer to an individual, group or organisation that applies components as a part of some software system, called host system. End-user refers to a corporate or individual user of the latter. Component developer or supplier refers to one developing and maintaining components.)

A critical difference for either type of component is whether the unit is regarded by customers as a black-box or a white-box element. The ensuing discussion focuses on the black-box case leaving the white-box situation for future analysis. Some questions addressing the relationship between CBSE and the laws follow.

- From the point of view of those who acquire components for integration into a system, what is the relevance of the laws?
- Is the widespread use of components likely to materially affect the validity of the laws as formulated, so as to invalidate them or require their substantial modification?
- What, if any, is the impact of component usage on the maintainability of the system within which they are integrated?
- Is it possible or even reasonable to expect to be able to ensure continued satisfactory operation of an E-type system in the changing domains, hard, soft and human, within which it operates and of which once deployed it becomes a part, when using components?

Before briefly addressing these questions (Sections 4 and 5) some properties of S- and E-type software, and a principle of uncertainty [22, 23] relating to the latter, are briefly discussed.

3 S- and E-type software

3.1 Basis of classification [8, 24]

When first formulated [15] the laws were related to large systems, to a concept of program largeness, as subsequently discussed [16, 17]. A universal and wholly satisfying definition of largeness was and remains elusive. The issue was clarified by a classification scheme distinguishing three types of programs S, P and E [18]. The laws were seen to apply to all E-type systems (as defined below) irrespective of their size, functional nature, domain of application, or the management structure and organisation responsible for them. The brief discussion that follows is restricted to types S and E since subsequent to the original formulation it was shown that type P can always be considered as satisfying the defining properties of one of the other two [25].

An S-type program is one that is accepted on the sole criterion that it satisfies a stated specification [18]; it must be verifiable to be acceptable. (Properties such as consistency, formality, completeness and satisfaction of the specification are discussed in [26, 27].) Validation to demonstrate that the program does not contain undesirable features not excluded by the specification is, in general, desirable but is excluded from the S-type definition [26].

An E-type program may, for simplicity, be defined as one whose acceptability depends on the perception, judgement and degree of satisfaction of appropriate stakeholders [18]. Software used to solve a problem or address an application in a real world domain is in general of this type. Such software is a model of a solution to some application in the domain of interest [18]. Both the application and its operational domain are intrinsically unbounded. No matter how many observations or properties are identified and associated with either, one can always add another. The software, on the other hand, is finite. The software system is therefore intrinsically incomplete. Moreover, the operational software is an intrinsic part of the domain and must contain an implicit model of itself. But a finite model cannot contain a model of itself. Every E-type system entails therefore multiple incompleteness. The resultant gap between the system and its operational domain is bridged by assumptions, explicit and implicit, by commission or omission. Reflections of these assumptions are embedded in the software.

The real world domain and the application itself are always changing. Installation and operation of the system may accelerate that change. Thus even supposing that the initial assumption set was valid, individual assumptions will, as time goes on, become invalid with unpredictable results or, at best, operation that is not totally satisfactory. This phenomenon may be seen as the ultimate source of the unending need for maintenance of E-type software, universally experienced since the beginning of the computer age. This analysis leads to the principle of uncertainty of computer application. This states that the outcome, in the real world, of software system operation is inherently uncertain [22, 23].

3.2 Are components of type S or E?

In isolation, complete and formally specified components satisfy the definition of S-type systems, with all other potential properties not included in the specification implicitly defined as don’t care. From the point of view of the user, their properties are defined by and only by their specifications. Provided that the specified interface definitions are completely adhered to, the unit will continue to behave as specified, when built (integrated) into some host system. However, the host will, sooner or later, require functional or nonfunctional modification or extension. Consequent changes to the system may require changes in component properties. If the required functional or behavioural changes are not implemented, the component will misbehave in execution. Thus, once integrated into a system, components must be judged by their behaviour and not by compliance with the original specification. Once integrated into the host, components become of type E.

If a component satisfies a precise specification, then while in isolation, ‘on-the-shelf’, the component is of type S. Once integrated into a real world application system, they display attributes of type E. Their life cycle and other behavioural characteristics must be expected to conform to
that encapsulated in the laws of software evolution. This is the key observation on which the discussion that follows is based. It implies that components too must be evolved. The remainder of this paper summarises the implications of this assertion.

4 Laws and their impact on component-based S.E.

4.1 Introductory remark
To facilitate the discussion, the laws, as recently reformulated [14], are individually stated and some of their implications briefly examined. Their formulation has been refined as further insight has been gained through the study of a number of systems developed by different organisations for a variety of applications in significantly different development environments [14]. But the precise form they take is not of major consequence for this discussion. It is now recognised that since the laws are not independent they should not be linearly ordered. Thus their numbering has only historical significance. Its use in structuring this paper is a matter of convenience and has no other implication. The question of dependencies and relationships between the laws is currently the subject of further investigation and an introductory description of that investigation is now available [9].

4.2 First and sixth laws
Continuing change (I): An E-type system that is used must be continually adapted else it becomes progressively less satisfactory.

Continuing growth (VI): The functional capability of an E-type system must be continually increased to maintain user satisfaction over its lifetime; where, in the context of the present paper, the term user applies to both component customers and end-users.

These two laws reflect distinct phenomena and different mechanisms. When phenomena are observed it is often difficult to determine which of the two laws underlies the observation. For the purpose of this paper they are considered together.

The first law relates primarily to the fact that the operational domain of any E-type software undergoes continuing change. Such change is partly driven by installation and operation of the system, partly by other forces, for example, human desire for improvement.

The sixth law is more related to the fact that finiteness of the implemented software implies that its properties are bounded relative to those of the application and its domain. Properties excluded by the bounds eventually become a source of performance limitations, irritation and error. To eliminate the latter requires extension of the system.

In selecting a component for integration into a system one must be clearly concerned with the adequacy of its behaviour after its integration into the operational system. This will be assessed on the basis of a functional specification, substantiated during the various process steps and finally confirmed during system validation. Precise data on interface, representational and linguistic characteristics will be of major concern in designing and implementing the system within which the component is embedded. Other functional and nonfunctional factors too will be of concern. The totality of factors considered must be bounded. The bounding process involves assumptions, explicit or implicit, conscious or unconscious, as indicated in Section 3.1. Integration of the component is likely to introduce further assumptions. The discussion that follows focuses on the issues raised by the inevitable presence of the latter.

The first and sixth laws imply that the application, the system and the domain undergo continuing change. This will often require modification of individual components to adapt them to the changed operational environment. This may be achieved by requesting the component developer to implement the necessary changes, by adapting the component software locally, by implementing more difficult, complex and obscure changes to the embedding system or the application, by obtaining a new component from an alternative source or by in-house implementation of a replacement. We illustrate the many possible scenarios by an example.

When component software is upgraded by an external development organisation serious delays may occur between identification of a need for modification or addition and its implementation. Moreover, there are likely to be conflicts between the needs and priorities of different users. Individual host organisations will still have to implement temporary patches to the component or the embedding system software, a self-defeating exercise both short term and long term and one likely to lead to the introduction of new defects. And if and when the component supplier does upgrade and supply the software at source, installation requires that patches be removed or modified and that the system be fully revalidated.

If, on the other hand, the necessary changes are achieved from within the customer organisation by the application of changes to the components and/or the code within which the components are embedded, the maintenance burden is likely to increase beyond that which would have been required had components not been used in the first place. The link with the component supplier, the support provided by the supplier and the link with other component customers using the same system will be weakened or severed altogether. De facto, the organisation has reverted to current practice of internal development with full local responsibility for future maintenance. This may represent a significant negation of the justification that led to selection of the component software in the first place.

In the face of a need for adaptation, and if the initial benefits of using components are to be preserved, the most reasonable choice is to stay with the original component code. The first and sixth laws, however, suggest that this may not be feasible. The introduction of CBSE is unlikely to reduce the need for system evolution, hence, if such evolution is not properly managed it is likely to involve an increased maintenance burden and consequently to introduce additional quality, reliability and maintenance issues.

This example reflects the phenomenon that underlies the principle of uncertainty (Section 3.1), that assumptions embedded in components progressively become invalid. If this was true for systems such as those involved in the sinking of HMS Sheffield, Airbus 320 crashes, the destruction of Ariane 501 [28], the Y2K bug and so on, how much more so for component-based systems. (It is, unfortunately, not possible to go into detail here but underlying these and similar incidents, past and future, lie assumptions, initially justified, that ultimately became or will become invalid.) Component developers cannot be aware of all assumptions to be made in the future by customers. Nor can the latter be aware of all assumptions reflected in a component item unless these are fully documented. The latter is, of course, impossible, if only because some, at least, of the implicit assumptions relate to the operational domain in which the item is used. Thus the use of
components, while initially attractive, is in the long run likely to prove burdensome, unreliable, expensive and a security risk. It is, however, possible that appropriate preventive measures as indicated in Section 5 may reduce the severity of such penalties.

4.3 Second law
Increasing complexity (II): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

An isolated component may satisfy the properties of S-type software as outlined in Section 3.1. Each new component release developed in response to customer requests and/or component developer self-interest, must be regarded as a new program. The members of a series of such releases successively integrated into customer systems become type E. They will be seen to follow the law since they must be adapted to satisfy a mix of changed requirements from a variety of sources with independent, possibly orthogonal, needs and assumptions. Unless work is done to control component complexity, the resultant increase will be reflected in a growing maintenance burden leading to a decline in maintenance productivity, as discussed in Section 4.2.

Complexity growth due to the volume and likely orthogonal nature of successive changes means that the component supplier may find it increasingly difficult and expensive to respond to component user needs in timely fashion with a high quality product. In particular, once a component is marketed, the supplier may not have sufficient incentives to invest significant effort in complexity reduction and other clean-up or re-engineering even if the cost is shared by its customers. In the face of market and competitive pressures and with a focus on short-term profits, what will become the accepted practice, remains to be seen. For example, in a market dominated by component suppliers, the latter may prefer to invest in the refinement and extension of their products to attract new customers, rather than in cleaning-up existing products. Indeed, circumstances may force suppliers to withdraw support from individual components and offer a replacement which, despite a claim of upward compatibility, will inevitably involve new assumptions that may conflict with the host system. A whole series of problems will emerge as both the component based system and the components on which it relies are required to evolve. Whatever evolution strategy is adopted by component suppliers, it will have an impact on their customers.

Assessing the long term benefit or penalty of using components is a major challenge, partly based on facts not presently available.

It may be that the introduction of components rather than the use of traditional development, will lead to faster and more effective preparation of the initial version of a host system. Moreover, the introduction of components will apparently reduce the constraints implied by the second law. However, this, and other factors (Section 4.1), will lead to an increase in the functionality and size of software systems. As an increasing number of components are used, each is effectively a primitive in a language defined by the full set of component units, with the interfaces providing the syntax of their use. Combining these observations one must conclude that the behaviour and constraints implied by the second law will eventually reappear, though at a higher level of abstraction. In the long term the constraints implied by the law are difficult to avoid.

4.4 Third, fourth and seventh laws
Self regulation (III): Global E-type system evolution processes are self regulating. (The global process includes all activities that influence the software process and its product, and includes not only direct technical activity but also that of other stakeholders such as business executives, marketeers, users and their managers.)

Conservation of organisational stability (invariant work rate) (IV): Average global activity rate in an E-type process tends to remain constant over periods or segments of system evolution.

These laws imply constraints on process behaviour and performance. In the context of component development these laws are likely to be relevant to both suppliers and users since they are closely bound up with the respective processes as determined by local and interconnecting feedback loops [14]. Supplier and customer development environments are likely to differ significantly but in the perspective of the global evolution process, the net impact is unlikely to change dramatically as a result of component usage. Detailed analysis of this hypothesis is a matter for further investigation.

4.5 Fifth law
Conservation of familiarity (V): The average growth rate of E-type systems tends to remain constant or to decline.

Originally this law [17] stated that the average incremental growth of a software system remains constant from release to release. This despite the fact that improving technology, improving skills and a growing familiarity of both developers and users with the application and the system could be expected to lead to an increasing growth rate. As an E-type system evolves, however, satisfactory usage and further evolution demands that both users and developers must maintain mastery of its content and behaviour. Thus, after each major release, established familiarity with the application and system in general is counterbalanced, at least temporarily, by a decline in the detailed knowledge and mastery of the system. This would be expected to produce a temporary slow down in the system growth rate or even system shrinkage as it is recognised that the system must be cleaned up to simplify the process of refamiliarisation. It is also compatible with an earlier claim that incremental system growth surpassing some threshold tends to be followed by one or more smaller or even negative increments [17].

As the system evolves, the complexity and orthogonality of changes from earlier concepts and system architecture will certainly increase the difficulty of making changes. The inevitable turnover of maintenance personnel, the introduction of fresh and relatively less experienced personnel who are less familiar with the system, its concepts, architectural principles and the assumptions that underlie it will, directly and by increasing the defect rate, so contributing to increasing difficulty in achieving satisfactory evolution.

FEAST observations suggest that pressures tending to decrease the growth rate dominate over pressures for increasing growth rates such as those arising from human ambition, competitive pressures and increasing familiarity with the application, system and their domains [20]. This observation is consistent with the inverse square growth trend observed in the projects [14, 29].

As stated, the law does not have direct implications for component users other than that it limits the rate at which a supplier can respond to user needs and requests. It also suggests that excessive demands in terms of needs or the
timescale within which these must be satisfied, even if apparently satisfied, will subsequently lead to a decline in quality, an increase in fault rates and delays in satisfying subsequent requests.

4.6 Seventh law
Declining quality (VII): Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
Assuming quality to be related to the extent to which system behaviour addresses the needs of an application and its domain, the law derives from the observation that user needs inevitably change with time. In the first place, therefore, the law relates to the host system. Nevertheless, it also impacts component suppliers in that it is a source for the demand for component maintenance and evolution. Both suppliers and customers will have to share the burden as they cope with the increasing difficulties and costs incurred in responding to end-user needs, by seeking to maintain high quality products in timely fashion.

4.7 Eighth law
Feedback system (VIII): E-type software processes are multilevel, multiloop, multigent feedback systems.
This statement is difficult to deny. Its implications and the potential the law offers for process improvement were explored in the FEAST projects [14]. The statement identifies the nature of the phenomenon and, thereby, the source of the observed behaviour encapsulated in the other seven laws. Its relevance to the use of components relates to the design and improvement of software processes which involve, or are intended to involve, components.

An intrinsic property of a complex feedback system is that it develops a dynamics of its own [14]. As a result of the integration of its products, the component development organisation becomes part of that larger system. The dynamics of its software and business process will influence that of the overall system and so impact the component customers and end-users. How? To ensure future support of the component incorporated into their host system, the component users will have little choice but to accept releases with changes not required by the system or even ones that are undesirable to them. Required changes may be delayed or even denied due to the component developers’ preoccupation with work for other customers or with activities directed to support their own or the component suppliers’ business or other interests. Changes in technology [3] or other aspects affecting the component supplier organisation directly will also feed through to the customer.

All these interactions represent feedback loops and system dynamic [30, 31] influences that process personnel must take into account in planning, managing and attempting to improve the development and evolution of their software systems. Failure to do so will lead to the serious long-term quality and cost problems discussed.

5 Recommendations
5.1 Long-term component support
In response to market and various feedback forces, component suppliers will evolve their products. Adoption of a new component version demands comprehensive checking of all known assumptions, implicit or explicit, in the host system, in the change implementation and in both the original and replacement components. If new versions are not adopted the host organisation will eventually lose supplier support. Either way significant problems will arise. A minimal condition for component adoption is expressed willingness of the supplier to support host maintenance as required, to provide access to component source code or deposit of the source code in escrow, as suggested in [32]. It follows that when considering adoption of CBSE, organisations must negotiate an appropriate and binding relationship with the component developer/supplier. For in the maintenance and evolution context, the customer organisation becomes hostage to the latter.

5.2 Embedded assumptions
As host organisations develop, maintain and evolve their systems they must consider not only the declared specification of each component but also the impact of the assumptions embedded in their own system and in the components. This clearly involves a significant and difficult challenge. In particular, the suppliers will likely be unwilling, probably unable, to provide comprehensive data on the assumption set underlying the component other than those published in the specification. Clearly, they will be unable to provide information on implicit or unconscious assumptions or those resulting from omission. The dependence on assumptions embedded in components introduces a new level of complexity and uncertainty [22, 23] in development and more so in system maintenance and evolution and will, it is believed, in the future constitute a significant source of defects identified as the system is used.

5.3 Continuing validity of laws
The third major issue that has been briefly considered relates to the continuing validity of the laws and their implications on practical software evolution and management of component-based systems. This complex issue deserves further investigation. The empirical data required for such study can perhaps be obtained from case studies of industrial component-based software evolution.

5.4 Specific recommendations
Functional and interface definition of components is necessary but not sufficient. One major addition to current development practice can be suggested. Given the inevitable embedding in any system of large number of assumptions about the real world application being addressed and the operational domain in which it is applied. (There is reason to believe that one such assumption may be found for every ten or so lines of code though no verification by actual count is known to the authors. But even if in error by a factor of two or more, the resultant number of embedded assumptions is such that for any system other than a trivial one, a significant amount of continuing rework is needed to maintain validity of the assumption set and, therefore, user satisfaction.) It follows that such assumptions must be captured and recorded in readily accessible fashion. (Our view of assumptions bears some relationship with the conceptual COTS framework mentioned in [32]). They must be fully and freely available to both direct and indirect users, individual and corporate, and must be reviewed whenever a change to the system, to the domain or to application practice is made. None of these actions seems to be reflected in current industrial practice or in their process models.

Taking the capture, recording and regular review of assumptions seriously must become a high priority. Proce-
dures for monitoring and evolving them must be developed. Such procedures must record a mapping between assumptions and software elements. (That is, the different models of the application, including requirements, specifications, high-level and low-level designs, code [26], support tools and CASE tools. All contain embedded assumptions which may affect the characteristics of the final delivered software.) The parts affected by the invalidation of any assumption may then be rapidly identified, the consequences assessed and reliable corrective action implemented. Making developers more aware of the existence and implications of assumptions will bring immediate as well as long term pay off. It is an activity that must be adequately supported. Development of appropriate methods and tools will follow but will prove neither trivial nor proof, if only because so many assumptions (if not most) are implicit, or made on-the-fly by individuals.

In connection with this brief discussion of assumptions, the following recommendations emerge:

- Capture, record, structure and periodically review assumptions made throughout the evolution processes of both components and host systems.
- Give preference to components for which a precise specification against which the component has been validated is provided and which includes a comprehensive list of known assumptions and recognised don’t cares.
- Require assurances that as components are evolved, customers will be informed whenever assumptions are negated and will be provided with comprehensive documentation and provided with the appropriate support when reintegrating such components.
- Introduce process steps that address issues of the impact of embedded assumptions, both before adoption of component units and whenever changes are made.

We have concentrated on one family of recommendations. Relationships of the latter with the laws and further recommendations that apply to software evolution planning and management in general are discussed in [24]. Others are expected to emerge from ongoing and planned work [9].

6 Conclusions

The results of the FEAST investigation [14] into software implementations not based on components indicate that the phenomena encapsulated by the laws of software evolution are real and probably universal. This paper has discussed implication of the laws in the CBSE context and in doing so has provided a discussion from which one can derive hypotheses for future investigations, empirical and theoretical, of the evolutionary attributes of CBSE processes and their products. The discussion has been restricted to the case in which component developers and their customers belong to different organisations (e.g. usage of COTS).

It should be re-emphasised that the eighth, feedback, law is seen as basic. It implies the existence of feedback mechanisms. The other seven laws reflect the consequences of their operation. Accepting that the global process is a dynamic feedback system has many implications. The most wide ranging is, probably, the observation that, in managing their activities, development and maintenance organisations must concern themselves with the total system involved. In the context of component usage this implies, for example, that the dynamics of the component development and supply sources must be treated as an integral part of the global system dynamics. The implications of this and methods by which it can be achieved require and deserve further study. Solutions appear feasible but even partial solutions are still some way off.

The wider use of component, and the reuse of locally developed software components, is widely seen as a response to the problems of ab initio development of major software systems [1] and the universal need for neverending maintenance of installed software [10]. The reasoning applied here suggests that the use of components may provide short-term effectiveness but introduce long term problems, intensifying the very phenomenon it was intended to control and reduce.

There is evidence to suggest that forces outside the immediate technical evolution process drive and constrain software system evolution [14]. There are good reasons to believe that, for component-based software such external forces may be even stronger, introducing into the evolution situation such factors as economic, business and even technology changes, whether isolated or cyclic [3]. This poses an extra challenge and load on those in charge of planning, managing and executing evolution. Further studies are needed to fully master these issues and to provide tools and methods appropriate to this newly recognised situation. These tools are essential if mankind is to survive its ever-growing dependence on computers, that is on software.

Overlooking or ignoring phenomena that unquestionably exist, that have been recognised, that are increasingly being unified in a body of knowledge and understanding about the software process [9], can only be harmful. Given full understanding of the nature of the problem, appreciation of its consequences, development and application of appropriate methods and technologies and adequate precautions, it would appear certain that the widespread use of components and the trend to reuse software will eventually prove advantageous. The benefit resulting therefrom will exceed any risks or negative effects that such practice can bring with it.

In the present paper we have only covered the fringe of an emerging topic. We have sought to draw the attention to the inevitability of the evolution of computer applications and of the software that implements them and of some of the implications of that evolution. Other papers, of ours [14] and of other authors [10, 11, 32], discussed different aspects of evolution and of software component usage. The whole area raises many issues, technical and business wise, and deserves careful attention. One thing is clear. For the moment the message must be ‘Proceed with caution, forget the panacea’.

7 Acknowledgments

We are grateful to the other members of the FEAST/2 team, Dr. Goel Kahn and Ms. Siew F Lim, and to Professors W. Turski and D.E. Perry. Thanks are due to the FEAST industrial collaborators for having provided data, interpretations and participation in many useful discussions. Thanks are also due to the anonymous referees for their comments and suggestions. We acknowledge financial support from the UK EPSRC, grants GR/K86008 (FEAST/1) and GR/M44101 (FEAST/2).

8 References

2 D’SOUZA, D., and WILLS, A.C.: 'Objects, components, and frameworks with UML: The catalysts approach' (Addison-Wesley, Boston, MA, 1999)
14 Feedback, evolution and software technology’, Project website, http://www.doc.ic.ac.uk/~mmill/feast

[Note: 1] [15–18] are reprinted as book chapters in [8].
[Note: 2] Also to appear in a proceedings to be published by the IEEE Computer Society.

IEE Proc.-Softw., Vol. 147, No. 6, December 2000