An Architectural Model for Service-Based Software with Ultra Rapid Evolution.

Keith Bennett, Malcolm Munro
Department of Computer Science
University of Durham, UK
keith.bennett@durham.ac.uk

Nicolas Gold, Paul Layzell*
Department of Computation
UMIST, UK
paul.layzell@umist.ac.uk

David Budgen, Pearl Brereton
Department of Computer Science
Keele University, UK
db@cs.keele.ac.uk

* address for correspondence

Keywords: software evolution, service architectures, process models

Abstract

There is an urgent industrial need for new approaches to software evolution that will lead to far faster implementation of software changes. For the past 40 years, the techniques, processes and methods of software development have been dominated by supply-side issues, and as a result the software industry is oriented towards developers rather than users. Existing software maintenance processes are simply too slow to meet the needs of many businesses. To achieve the levels of functionality, flexibility and time to market of changes and updates required by users, a radical shift is required in the development of software, with a more demand-centric view leading to software which will be delivered as a service, within the framework of an open marketplace. Although there are some signs that this approach is being adopted by industry, it is in a very limited and restricted form.

We summarise research that has resulted in a long-term strategic view of software engineering innovation. Based on this foundation, we describe more recent work that has resulted in an innovative demand-led model for the future of software. We describe a service architecture in which components may be bound instantly, just at the time they are needed and then the binding may be disengaged. Such ultra late binding requires that many non-functional attributes of the software are capable of automatic negotiation and resolution. Some of these attributes have been demonstrated and amplified through a prototype implementation based on existing and available technology.

1. Objectives

Software maintenance has matured considerably over the past 20 years so that standards representing best practice are emerging [1, 2]. Most approaches to software maintenance derive from three basic process stages [10]:

i) Understanding the existing software
ii) Modification of the existing software
iii) Revalidation of the modified software

In industrial practice, these three basic stages may expand to incorporate many sub-stages, with audits, and control boards taking on supervisory responsibility. Empirically, this process has been successful in supporting the maintenance of large, risk-averse software systems (such as safety critical and business critical systems). Although the "applications backlog" has always been recognised as a problem, users have become familiar with the activity of new releases of software (perhaps every six to 24 months), in which groups of change requests, enhancements and/or bug fixes have been aggregated and made available by vendors. Procedures for "emergency fixes" can be used to resolve urgent problems, though these are a well known source of later difficulty. Even in very mature software maintenance processes, the skills of the maintainers and their system knowledge are usually critical.

In [5], a challenge was made to the conventional, categorised view of software maintenance (adaptive, perfective, corrective, preventative, etc.). Instead, it was argued that maintenance should be categorised, within a staged model, according to the phase of the

maintenance lifecycle. Five main stages were identified:

- **Initial development** - the first functioning version of the system is developed.
- **Evolution** - the engineers extend the capabilities and functionality to meet the needs of its users, possibly in major ways.
- **Servicing** - the software is subjected to minor defect repairs and simple changes in function.
- **Phase out** - no more servicing is being undertaken, and the owners seek to generate revenue from the use for as long as possible.
- **Close down** - the software is withdrawn from the market, and any users directed to a replacement system if this exists.

This is a more useful model with which to address the needs of many modern businesses. The internet age has ushered in a new era of highly dynamic and agile organisations which must be in a constant state of evolution if they are to compete and survive in an increasingly global marketplace. These are operating in a time-critical environment, rather than a safety critical application domain. If a change or enhancement to software is not brought to market sufficiently quickly, thus retaining competitive advantage, the organisation may collapse. This era poses significantly new problems for software development, characterised by a shift in emphasis from producing 'a system' to the need to produce 'a family of systems', with each system being an evolution from a previous version, developed and deployed in shorter and shorter business cycles. In other words, the evolution stage in the above model becomes central. It may be that the released new version is not complete, and still has errors. If the product succeeds, it can be put on an "emergency life support" to resolve these. If it misses the market time slot, it probably will not succeed at all.

The traditional software maintenance process may have release intervals of months or even years when a very large software system is being evolved. This is far too long for many organisations; they require evolution in days or even hours (evolution in so-called "internet time").

It is possible to inspect each activity of the software evolution process and determine how it may be speeded up. Certainly, new technology to automate parts may be expected, supported by tools (for example, in program comprehension, testing etc.). However, it is very difficult to see that such improvements will lead to a *radical* reduction in the time to evolve a large software system. This prompted us to believe that a new and different way is needed to achieve "ultra rapid evolution"; we term this "evolution in internet time". It is important to stress that such ultra rapid evolution does not imply poor quality, or software which is simply hacked together without thought. The real challenge is to achieve very fast change yet provide very high quality software. Strategically, we plan to achieve this by bringing the evolution process much closer to the business process.

In 1995, British Telecommunications plc (BT) recognised the need to undertake long-term research leading to different, and possibly radical, ways in which to develop software for the future. Senior academics from UMIST, Keele University and the University of Durham, came together with staff at BT to form DiCE (The Distributed Centre of Excellence in Software Engineering). The outcome of this research is summarised in Section 2 of the paper. In Section 3, we express the objectives of the current phase of research in terms of the vision for software -how it will behave, be structured and developed in the future. From 1998, the core group of researchers switched to developing a new overall paradigm for software engineering: a service-based approach to structuring, developing and deploying software. This new approach is described in the second half of this paper. In section 4, we describe a prototype implementation of the service architecture, demonstrating its feasibility and enabling us to elucidate research priorities.

2. Developing a Future Vision

The method by which the DiCE group undertook its research is described in [4]. Basically, the group formulated three questions about the future of software: How will software be used? How will software behave? How will software be developed? In answering these questions, a number of key issues emerged.

K1. Software will need to be developed to meet **necessary and sufficient requirements**, i.e. for the majority of users whilst there will be a minimum set of requirements software must meet, over-engineered systems with redundant functionality are not required.

K2. Software will be **personalised**. Software will be capable of personalisation, providing users with their own tailored, unique working environment which is best suited to their personal needs and working styles,

thus meeting the goal of software which will meet necessary and sufficient requirements.

K3. Software will be **self-adapting**. Software will contain reflective processes which monitor and understand how it is being used and will identify and implement ways in which it can change in order to better meet user requirements, interface styles and patterns of working.

K4. Software will be **fine-grained**. Future software will be structured in small simple units which co-operate through rich communication structures and information gathering. This will provide a high degree of resilience against failure in part of the software network and allow software to re-negotiate use of alternatives in order to facilitate self-adaptation and personalisation.

K5. Software will operate in a **transparent** manner. Software may continue to be seen as a single abstract object even when distributed across different platforms and geographical locations. This is an essential property if software is to be able to reconfigure itself and substitute one component or network of components for another without user or professional intervention.

Rapid evolution interacts strongly with these demands, and hence a solution which had the potential to address all the above factors was sought.

3. Service-Based Architecture

3.1 The Problem

Most software engineering techniques, including those of software maintenance, are conventional supply-side methods, driven by technological advance. This works well for systems with rigid boundaries of concern such as embedded systems. It breaks down for applications where system boundaries are not fixed and are subject to constant urgent change. These applications are typically found in **emergent organisations**-- "organisations in a state of continual process change, never arriving, always in transition" [6]. Examples are e-businesses or more traditional companies who continually need to reinvent themselves to gain competitive advantage [7]. These applications are, in Lehman's terms, "E-type" [9]; the introduction of software into an organisation changes the work practices of that organisation, so the original requirements of the software change. It is not viable to identify a closed set of requirements; these will be forever changing and many will be tacit.

Subsequent research by DiCE has taken a **demand-led** approach to the provision of software services, addressing delivery mechanisms and processes which, when embedded in emergent organisations, give a software solution in emergent terms -- one with continual change. The solution never ends and neither does the provision of software. This is most accurately termed *engineering for emergent solutions*.

For software evolution, it is useful to categorise contributory factors into those which can rapidly evolve, and those which cannot:

Fast moving	**Slow moving**
Software requirements	Software functionality
Marketplaces	Skills bases
Organisations	Standards
Emergent companies	Companies with rigid boundaries
Demand led	Supply led
Competitive pressures	Long term contracts
Supply chain delivery	Software technology
Risk taking	Risk averse
New Business processes	Software Process evolution
Near-business software	Software infrastructure

Table 1: Evolutionary Rates of Various Contributing Factors

We concluded that a "silver bullet", which would somehow transform software into something which could be changed (or could change itself) far more quickly than at present, was not viable. Instead, we took the view that software is actually hard to change, and this takes time to accomplish. We needed to look for other solutions.

3.2 Service Architecture

Currently, almost all commercial software is sold on the basis of ownership (we exclude free software and open source software). Thus an organisation buys the object code, with some form of licence to use it. Any updates, however important to the purchaser, are the responsibility of the vendor. Any attempt by the user to modify the software is likely to invalidate warranties as

well as ongoing support. In effect, the software is a black box that cannot be altered in any way, apart from built-in parameterisation. This form of marketing is known as supply-led. It is the same whether the software is run on the client machine or on a remote server, or, if the user takes on responsibility for in-house support or uses an applications service supplier (i.e. outsources maintenance).

Let us now consider a very different scenario. We assume that our software is structured into a large number of small components (see K1, K4, K5 above), which exactly meet the user's needs and no more. Suppose now that a user requires an improved component C. The traditional approach would be to raise a change request with the vendor of the software, and wait for several months for this to be (possibly) implemented, and the modified component integrated.

In our solution, the user disengages component C, and searches the marketplace for a replacement C' which meets the new needs. When this is found, it is bound in instead of C, and used in the execution of the application. Of course, this assumes that the marketplace can provide the desired component. However, it is a well established property of marketplaces that they can spot trends, and make new products available when they are needed. The rewards for doing so are very strong and the penalties for not doing so are severe. Note that any particular component supplier can (and probably will) use traditional software maintenance techniques to evolve their components. The new dimension is that they must work within a demand-led marketplace. Therefore, if we can find ways to disengage an existing component and bind in a new one (with enhanced functionality and other attributes) ultra rapidly, we have the potential to achieve ultra-rapid evolution in the target system.

This concept led us to conclude that the fundamental problem with slow evolution was a result of software which is marketed as a product, in a supply-led marketplace. By removing the concept of ownership, we have instead a service i.e. something which is used, not owned. Thus we generalised the component-based solution to the much more generic service based software in a demand led marketplace.

This **service-based model of software** is one in which services are configured to meet a specific set of requirements at a point in time, executed and disengaged - the vision of instant service, conforming to the widely accepted definition of a service:

> "an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production" [8].

Services are composed out of smaller ones (and so on recursively), procured and paid for on demand. A service is not a mechanised process; it involves humans managing supplier-consumer relationships. This is a radically new industry model, which could function within markets ranging from a genuine open market (requiring software functional equivalence) to a *keisetzu* market, where there is only one supplier and consumer, both working together with access to each other's information systems to optimise the service to each other.

This strategy enables users to create, compose and assemble a service by bringing together a number of suppliers to meet needs at a specific point in time. An analogy is selling cars: today manufacturers do not sell cars from a pre-manufactured stock with given colour schemes, features etc.; instead customers configure their desired car from series of options and only then is the final product assembled. This is only possible because the technology of production has advanced to a state where assembly of the final car can be undertaken sufficiently quickly.

Software vendors attempt to offer a similar model of provision by offering products with a series of configurable options. However this offers extremely limited flexibility -- consumers are not free to substitute functions with those from another supplier since the software is subject to *binding* which configures and links the component parts, making it very difficult to perform substitution. The aim of this research is to develop the technology which will enable *binding* to be delayed until immediately before the point of execution of a system. This will enable consumers to select the most appropriate combination of services required at any point in time.

However *late binding* comes at a price, and for many consumers, issues of reliability, security, cost and convenience may mean that they prefer to enter into contractual agreements to have *some early binding* for critical or stable parts of a system, leaving more volatile functions to late binding and thereby maximising competitive advantage. The consequence is that any

future approach to software development must be interdisciplinary so that non-technical issues, such as supply contracts, terms and conditions, and error recovery are addressed and built in to the new technology.

3.3 Bind Once, Execute Once

A truly service-based role for software is far more radical than current approaches, in that it seeks to change the very nature of software. To meet users' needs of evolution, flexibility and personalisation, an open market-place framework is necessary in which the most appropriate versions of software products come together, are bound and executed as and when needed. At the extreme, the binding that takes place prior to execution, is disengaged immediately after execution in order to permit the 'system' to evolve for the next point of execution. Flexibility and personalisation are achieved through a variety of service providers offering functionality through a competitive market-place, with each software provision being accompanied by explicit properties of concern for binding (e.g. dependability, performance, quality, licence details etc).

A component is simply a reusable software executable. Our *serviceware* clearly includes the software itself, but in addition has many non-functional attributes, such as cost and payment, trust, brand allegiance, legal status and redress, security and so on. Binding requires us to negotiate across all such attributes (as far as possibly electronically) to establish a binding, at the extreme just before execution.

4. Service Implementation – Prototype and Results

4.1 Aims

The main aims of this and future prototypes are to test ideas about the structure and operation of the software service architecture, and to illuminate problem areas for further research. At present, these are very much proof-of-concept tools to clarify our thinking on certain aspects of the problem – in this case, ultra-late binding.

4.2 Problem Addressed

The first prototype was designed to supply a basic calculation service to an end-user. The particular calculation selected was the problem of cubing a number. Note that due to the service nature of the architecture, we aim to supply the *service* of cubing, rather than the *product* of a calculator with that function in it. This apparently simple application was chosen as it highlights many pertinent issues yet the domain is understood by all.

4.3 Entities Involved

Three main entities are involved in service delivery in the prototype: the end-user, an interface, and service providers. Figure 1 shows the relationships between them and Table 2 summarises the process of service provision. The interface (in this case, a web browser) allows the end-user to interact with delivered software. It is expected that the interface will be light-weight and perhaps supplied free in a similar manner to today's web browsers.

Service providers are divided into three types:

- **Information service providers (ISPs)**: those that provide information to other services e.g. catalogue and ontology services.
- **Contractor service providers (CSPs)**: those that have the ability to negotiate and assemble the necessary components/services to deliver a service to the end-user.
- **Software service providers (SSPs)**: those that provide either the operational software components/services themselves, or descriptions of the components required and how they should be assembled.

4.4 Service Provision in the Prototype

Service from the end-user's point of view is provided using the following basic model:

1) The end-user requests a software service.
2) The end-user selects a service domain (e.g. calculation).
3) The end-user selects a service within the domain (e.g. cube).
4) The end-user enters the number they want to cube.
5) The end-user receives the result.

Apart from the notion of requesting the service of cube rather than the product of calculator, it can be seen that the process of cubing is similar to selecting the function from a menu in a software product. However, the hidden activity for service provision is considerable.

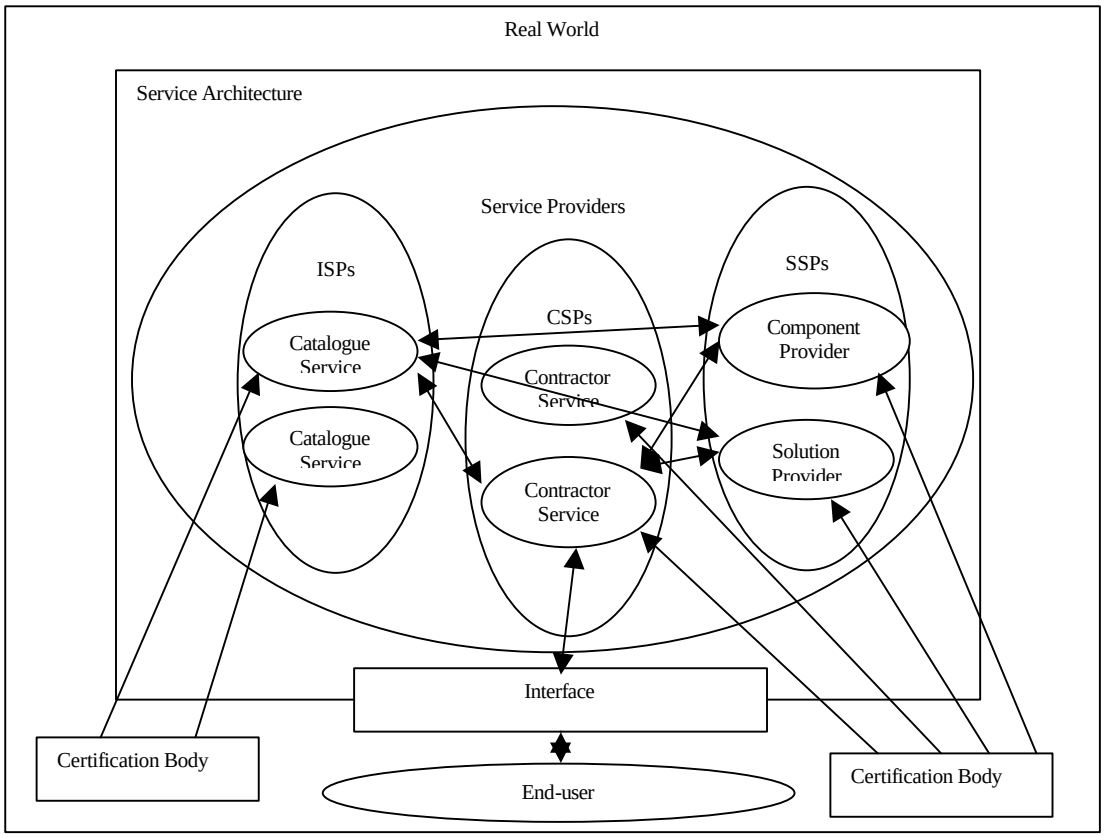

Figure 1: Service Architecture

Entity	Requires	From	Service Provided	Delivers	To	Examples
End-user				Software service request	Interface	"I want to cube this number"
Interface	Software service request	End-user	Negotiates a CSP and handles information transfer between customer and CSP.	Software service	End-user	Internet Browser
Contractor Service Provider	Software service request	Interface	Negotiation, binding, and execution of the software service	Software service requested	Interface	
Information Service Provider	Type of information	CSP	Lookup and filtering of all available information in accordance with search criteria	Information of the type required	CSP	Catalogue service Ontology service
Software Service Provider	Completed Contract	CSP	Supply of a software solution or supply/execution of a software component	Software solution or software component or results of executing a software component	CSP	

Table 2: Entities and Processes of Service Provision

We assume that the difficulties of obtaining a common understanding between end-user and service supplier(s) of the requirements will be handled through the use of a standard ontology. These are emerging with the development of web service technologies such as UDDI. Using a standard ontology removes the need for activities such as formal specification and matching.

Each provision of service is governed by a simple *contract.* This contains the terms agreed by the service provider and service consumer for the supply of the service. The specific elements of a contract are not prescribed in terms of the general architecture; providers and consumers may add any term they wish to the negotiation. However, for the prototype, three terms are required:

1) The **law** under which the contract is made.
2) Minimum **performance (**represented in the prototype by a single integer).
3) **Cost (**represented by a single integer).

In order to negotiate a contract, both end-users and service providers must define *profiles* that contain acceptable values for contract terms. The profiles also contain *policies* to govern how these values may be negotiated. The profiles used in the first demonstrator are extremely simple. End-user profiles contain acceptable legal systems for contracts, the minimum service performance required, the maximum acceptable cost, and the percentage of average market cost within which negotiation is possible. Service provider profiles contain acceptable legal systems for contracts, guaranteed performance levels, and the cost of providing the service. Negotiation in the prototype thus becomes a process of ensuring that both parties can agree a legal system and that the service performance meets the minimum required by the end-user. If successful, service providers are picked on the basis of lowest cost. Acceptable costs are determined by taking the mean of all service costs on the network for the service in question and ensuring that the cost of the service offered is less than the mean plus the percentage specified in the end-user profile. It must also be less than the absolute maximum cost.

To avoid the overhead of negotiation for basic, common services such as catalogues, it is assumed that external bodies will provide "certificates" which represent fixed cost, fixed performance contracts that do not require negotiation. Both end-user and service provider profiles contain acceptable certificates.

4.5 An Example

The example presented above from the end-user's viewpoint will now be repeated but with a description of the intermediate steps taken by the interface and service providers.

Service provision begins with the end-user recognising a need for calculation e.g. " I want to cube this number". The end-user starts the interface and requests software service (step 1 above). The interface uses a certificate to engage a catalogue service and requests a catalogue of contractor services currently available on the network. It contacts each of these services in the supplied catalogue and attempts to negotiate a contract with each using the end-user's profile. The best contract is selected and the requirements and end-user profile passed to the chosen contractor.

The contractor now acts on behalf of the end-user (thus concentrating negotiation and composition expertise in a service, not in the interface) and begins by engaging a catalogue service using a certificate to determine available solution domains. Having retrieved a list of domains, the contractor sends the domain list to the interface and the end-user is presented with a choice, selecting calculation (step 2 above). The contractor then engages a catalogue service for solutions within the calculation domain. Again, a list is presented and the end-user selects cube (step 3 above). At this point, the contractor negotiates the supply of a cube solution description from a software service provider. The solution description tells the contractor service which other services are required to perform cubing and how to compose them. The contractor then uses catalogue services to find these supporting services, negotiates contracts with them, and supplies them in the correct order to the interface for the end-user to use. The description of a cube service might be input1|cube|output1. This is a pipeline where the first service is capable of receiving a number from the end-user (step 4 above), the second cubes it, and third outputs a number (i.e. the result, step 5 above). No restriction is placed on by whom these services are provided, or where the software is executed.

Having completed the service provision, the contractor disengages all the supporting services, and returns control to the interface. If the user were now to cube another number, the process is repeated, but the market place may have changed; there may be faster and cheaper (sub)services available to undertake the service. So such a better service will be bound in just before execution. Thus different

services are bound in for successive uses of the cube service.

4.6 Implementation Technology

The prototype is implemented using an HTML interface in a web browser. PHP scripts are used to perform negotiation and service composition by opening URLs to subsidiary scripts. Each script contains generic functionality, loading its "personality" from a MySQL database as it starts. This allows a single script to be used to represent many service providers. End-user and service provider profiles are stored on the database, which also simulates a service discovery environment.

4.7 Prototype Results and Conclusions

The basic requirement for our solution to ultra-rapid evolution is very late binding, and subsequent disengagement. The prototype has demonstrated that the basic concept of a software-service architecture is feasible, and shows the basic primitives of the architecture are viable. A very simple application domain example has been sufficiently rich to enable demonstration of many of the basic ideas. Using simple scripts, some inter-service negotiation can be undertaken successfully to supply a cube service (comprised of sub services) to the end-user. The prototype has also been extended with little effort to supply an "addition" service. The implementation has relied almost completely on scripts, and has shown that a service architecture can, with very few restrictions, allow different negotiation, discovery and description methods (so the architecture is not prescriptive). This experimental work has provided three areas of evidence to support our aim of ultra rapid evolution:

- Very late binding, and subsequent disengagement can be achieved for both functional and non-functional service attributes, given suitable discovery, description and negotiation representations.
- The service architecture is not committed to particular description notations or negotiation mechanisms.
- The "leaves" of the supply chain are conventional software, evolved and supported using conventional well understood techniques.

Future prototypes may be able to take advantage of discovery and description technologies such as UDDI, WSDL, etc. further to demonstrate the success of the approach using emerging industry standards for B2B e-commerce.

5. Future Research Issues

Using the results of the first prototype, we have identified a number of major issues that need to be addressed.

Requirements for software need to be represented in such a way that an appropriate service can be *discovered* on the network. The requirements must convey therefore both the description and intention of the desired service. Given the highly dynamic nature of software supplied as a service, the maintainability of the requirements representation becomes an important consideration. However, the aim of the architecture is not to prescribe such representation, but support whatever conventions users and service suppliers prefer.

Automated *negotiation* is another key issue for research, particularly in areas where non-numeric terms are used e.g. legal clauses. Such clauses do not lend themselves to offer/counter-offer and similar approaches. In relation to this, the structure and definition of profiles and terms needs much work, particularly where terms are related in some way (e.g. performance and cost). Also we need insight to the issue of when to select a service and when to enter negotiations for a service. It is in this area that multi-disciplinary research is planned. We plan to concentrate research in these areas, and use as far as possible available commercial products for the software infrastructure.

Payment issues are clearly important in this framework and will need to be addressed. A range of possibilities exist from a once-only small payment for a single service provision, to an up front large payment for a number of provisions (e.g. for a discount). We anticipate the payment structure being built into the contract and negotiated with the other terms. Payment management might be handled using another service.

Finally, many issues need to be resolved concerning mutual performance monitoring and claims of legal redress should they arise.

6. Conclusions

We have presented a radical and innovative architectural approach to achieving ultra rapid evolution, which is one of five key requirements identified for 21st century software. This still assumes a software maintenance approach and technology based on that used currently. However, it postulates a completely different software marketplace, which is demand-led, not supply-led. Software is marketed as

a service which is used, not owned. A consumer, or supply chain service vendor integrates a number of such services to provide added value. However, the supply services are bound just before execution and disengaged afterwards, so a service can be replaced by an improved one when needed. By binding in such services, not as now when software is bought, but when it is executed, the software may be continually adapted to meet user requirements.

Of course, this raises many problems, and to start to answer these, a prototype service architecture has been built using current technology. Binding to a service right at the last minute requires automatic resolution, not only of functional attributes, but also (perhaps mainly) non-functional aspects too. Also we cannot impose a grand universal scheme on service suppliers and users; in a marketplace, each must be free to adopt their own ways of business.

The prototype has given us some insight into the issues involved in specifying, negotiating, and delivering service-based software. It has confirmed that the basic concept of software as a service is feasible, and has highlighted areas for future research.

Acknowledgements

We acknowledge the support of the Leverhulme Trust, British Telecom plc, and the UK Engineering and Physical Sciences Research Council.

References

[1] *IEEE Standard for Software Maintenance* (IEEE Std 1219-1998) in IEEE Standards, Software Engineering, Vol 2, Process Standards, 1999 Edition, IEEE 1999. ISBN 0738115606

[2] *International Standard: Information Technology - Software Maintenance* ISO/IEC 14764:1999

[3] Bennett K. H.,.Layzell P. J., Budgen D., Brereton O. P., Macaulay L., Munro M., *Service-Based Software: The Future for Flexible Software*, IEEE APSEC2000, The Asia-Pacific Software Engineering Conference, 5-8 December 2000, Singapore, IEEE Computer Society Press, 2000.

[4] Bennett K. H., Munro M., Brereton O. P., Budgen D., Layzell P. J., Macaulay L., Griffiths D. G. & Stannet C. *The future of software.* Comm. ACM, vol.42, no. 12, Dec. 1999, pp. 78 – 84.

[5] Bennett K. H. and Rajlich V. T. *A staged mode for the software lifecycle.* IEEE Computer, vol. 33, no. 7, pp. 66 –71, July 2000, ISSN 0018-9162.

[6] Truex D., Baskeville R. and Klein H., *Growing Systems in Emergent Organizations*, Comm.ACM, Vol.42, No.8, August 1999

[7] Cusumano M. & Yoffe D., *Competing on Internet Time – Lessons from Netscape and its Battle with Microsoft*, Free Press (Simon & Schuster) 1998

[8] Lovelock C., Vandermerwe S., & Lewis B., *Services Marketing*, Prentice Hall Europe, 1996

[9] Lehman M. M. *Programs, lifecycles and the Laws of Software Evolution.* Proc. IEEE, vol. 68, no. 9, September 1980.

[10] Bennett K. H., Cornelius B. J., Munro M., Robson D. *Software Maintenance.* In "The Software Engineer's Reference Manual" (Ed. J. McDermid), published by Butterworth, ISBN 0-750-61040-9, 1990.