Achieving ultra rapid evolution using Service-based Software

Keith Bennett
Department of Computer Science
University of Durham, UK
+44 (191) 374 2632
keith.bennett@durham.ac.uk

Nicolas Gold
Department of Computation
UMIST, UK
+44 (161) 200 3385
n.e.gold@co.umist.ac.uk

ABSTRACT

There is an urgent industrial need for new approaches to software evolution that will lead to far faster implementation of software changes. 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 see ultra rapid evolution, in "internet time" as a grand challenge for software engineering.

In this position paper, we describe recent work that has resulted in an innovative demand-led model for the future of software. We describe a service architecture in which services 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 through two prototype implementations based on existing and available technology. The aim of the position paper is to contribute to the debate at ESEC by presenting a radical, market based view of software evolution which must take place in "internet time". The key underpinning theoretical idea is ultra-late binding, so that a service is engaged dynamically at the point in time it is needed. Hence, the sub services can evolve between usages.

Categories and Subject Descriptors

D.2.7 [**Management**]]: lifecycle – *maintainability, late binding.*

General Terms

Management, design.

Keywords

Service-based software, service architecture, evolution.

1. SERVICE-BASED ARCHITECTURE

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".

Examples are e-businesses or more traditional companies who continually need to reinvent themselves to gain competitive advantage.

Currently, almost all commercial software is sold on the basis of ownership. 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, 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 identify 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".

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.

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.

2. 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).

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.

3. SERVICE IMPLEMENTATION – PROTOTYPE AND RESULTS

3.1 First Prototype

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.

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 were 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.

3.2 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.

3.3 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.

4. REFERENCES

[1] 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.

[2] 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.

[3] 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.