Architecture-driven component reuse

P.A.V. Hall

Open University, Computing Department, Walton Hall, Milton Keynes MK7 6AA, UK

Abstract

The evolution of software reuse over the last 30 plus years is drawn upon to show current achievements, a stable model of components, an approach to reusing components based on architectures, an appreciation of human and organisational problems in reuse, accumulating evidence for the value of reuse approaches. The shortfalls of some current OO methods are pointed out. The significance and limitations of software reuse is indicated by situating it in the wider context of learning organisations and knowledge management, and by viewing architecture-driven reuse as routine design. The distinct role of patterns is indicated.

Keywords: Architecture-based component reuse; Learning organisations; Knowledge management; Components; Reuse; Patterns; Frameworks

1. Introduction

For the third time, we are now cycling around research and development about components and reuse. The first time was in the development of libraries of useful routines, of mathematical functions and Unix utilities, only we did not call it reuse then, we just got on and did it because it made good sense. The second time was in the real time community in publicly funded projects in both the US and Europe, with projects like CAMP seeing if there was a better way to build the software for missiles. The third time round is being driven by business needs for flexibility in the face of change, being picked up in current object-oriented (OO) methodologies. It is marked by the appearance of many books with “Reuse” or “Component” in their titles, giving the illusion of a maturity that is really only partly there. Will we make it in the next millennium? Maybe. But one thing we must do is reuse the experience of the earlier rounds in this latest round of R&D about components and reuse.

We start with a product focus. The whole reuse idea revolves around the component. However much many researchers protest otherwise, this ultimately means the reuse of code, for the objective is to identify a software need, and then circumvent the expensive process of design and implementation by obtaining an executable component that meets that need. In Section 2, we look at the very well-established model of a component, and explain how objects do not meet the requirements of being a component.

Components themselves are not enough, they fit into a process of routine design where the potential place of a component in a system is anticipated through reusable designs or architectures, often called frameworks. This is picked up in Section 3, together with a discussion of the irrelevance of library technologies and the implicit matching requirements.

We then move to a process and commercial focus. Reuse and components are advocated as a means of reducing production costs and timescales, to combat the rising demand for software far outstripping the supply of software engineers to produce it. Reuse achieves this, but to understand how and why, we need to take a much broader view of the software development process than is usual. All too often advocacy of reuse has seemed like a moral crusade against a profligate software industry. In Section 4, we question the current project focused view of the benefits of reuse, and then in Section 5, take a broader industrial and business view.

But all is not lost, salvation may just be a millennium away. Section 6 discusses what needs to be done to make it really happen. At last!

2. Components and their interconnection

Software components have been with us ever since the invention of the closed subroutine of assembler languages. Fragments of code that did useful things could be packaged together and executed, remembering their invocation context on entry and reinstating it at exit. They could be placed in libraries to be linked into new programmes as required. In 1968, McIlroy [25] pointed out that these could be viewed as software components. Later, these
become re-embodied as modules [26,30], and then in 1976, De Remer et al. [13] observed that there were two levels of software production, “in-the-small” programming of modules or components, and “in-the-large” connection of these modules together. These MILs (Module Interconnection Languages) were further developed by Tichy and others (see Goguen [18] and the survey by Prieto-Diaz and Neigbors [31]). They were picked up in the late 1970s by the DRACO project [16,27], the first software reuse project, focused on components at various levels of abstraction. From this a very clear idea of a component emerged as a coherent package of programme code in which there are number of well-defined interfaces provided by the component for delivery of its functions to the software that uses it. They are sometimes also called “sockets”. In turn, the component may require the services of other components, and these required interfaces or plugs would need to be defined as part of the component. The functions of the component are also specified in terms of the provided and required interfaces. This view of components and modules has become part of many software development methods, such as MASCOT [35]; see Fig. 1. An important point to note is that the interface to a component consists of many parts, and that the same interface (part) may occur in many places. Interfaces are first class elements.

Now components may not be that simple. As seen in Fig. 1, components may be recursive, with a component itself being decomposed into a number of interconnected sub-components. These sub-components may be defined locally, but may be instantiatted from outside, in which case how are they different from required components? Semantically not at all, but structurally it is critical—decomposition enables you to define a structure both to the design of components and the software using the components.

In MILs, a component may also be made generic through parameterisation—the component at the time of incorporation into a design (instantiation) may have a particular parameter set. The stereotypical example is a stack of undetermined elements, with these elements determined at instantiation as a stack of integers or stack of vectors or pictures, or whatever. These generic parameters have also the flavour of required components, but are they really? Generic parameters should rather be viewed as decomposition components, as above.

The important point to emphasise here is that this view of components has been established for nearly 20 years, and has not been bettered.

In 1986, Brad Cox [9] championed objects as the candidate software components, calling these “software integrated circuits”. In 1986 this was okay, but this view continues in widespread mistaken use today; for example Ian Graham in 1998 writes:

“a component is merely an object (specification, source code, or binary) that is published for reuse: the interface specifies exactly what it will do.” [19, p. 206]

This object-orientation view is mistaken because objects do not capture the richness required of a component—in particular interfaces are not decomposed and required interfaces are not made explicit. And we may need to treat a collection of collaborating objects together as a component, rather than a single object. The emergent standard for representing object-oriented designs is UML, the Unified Modeling Language, and is worth quoting [29]:

“The UML has integrated many disparate ideas, so this integration will accelerate the use of object-orientation. Component-based development is an approach worth mentioning. It is synergistic with traditional OO techniques. While reuse based on components is becoming increasingly widespread, this does not mean that component-based techniques will replace OO techniques. There are only subtle differences between the semantics of components and classes.”

(UML; vol. 1.3, pp. 1–15)

The “components” construct of UML continues the OO misunderstanding of components, and the better constructs for components within UML are packages or subsystems that permit collections of objects that are not themselves objects and include a rudimentary form of the pluggability points demanded by the component model above. UML and its proponents clearly still have some way to go.

Current interest in objects has arisen from business
objects, often thought of as the object incarnation of database entities. This does naturally lead to the Graham view, but Eeles and Sims [15] show how as we move to distributed systems in multi-level client server architectures, a single business object must be split into distributable objects which collaborate across the network and together should be viewed as the component.

Further, even this view is too simple—Eeles and Sims still focus on small components, atomic components even, but what we are really interested in is larger components for higher reuse pay-off. Big components for big benefits, as we shall see.

Is a spreadsheet an object? It clearly can be a component in some larger system.

3. Components in context

In 1987, Grady Booch [5] developed a library of low level Ada components which he sold commercially. A project in a Swedish telecommunications supplier accumulated Ada components, including the Booch library, gathering altogether something like 4000 general purpose small components. The benefits hoped for were simply not achieved, and the project was abandoned. In contrast, around 1990 the HP telecommunications facility in Grenoble managed to use components very effectively for telephone switching systems, but they only had about 100 components. The success stories given by Jacobson et al. [23] are primarily of this kind. That HP in Grenoble had succeeded was quite a revelation for it had been assumed that in order to be able to provide something useful you needed very large libraries.

Through this and other examples, it was realised that successful reuse strategies had to focus on a particular area of application, a domain, and collect only components which are definitely known to be useful in that domain. Useful components would be found through domain analysis introduced by the DRACO project, see Refs. [16,27,32]). By collecting only components known to be useful, this meant that collections of components would be small and, therefore, that library retrieval systems were just not necessary; with only 100 components you can remember what is there and choose the right one. At worst a small paper-based manual, perhaps like those published for electronic components, would suffice.

Our experience is that components alone are not enough; what is needed in addition is an understanding of how we will use the components in the domain in which it will be deployed. This can be done through focused (domain specific) libraries, like the mathematical subroutines that McIllroy had in mind. The object-oriented form of this is Class libraries, which may include further advice on using the components to form a Framework. A framework can be as loose as a Class library, or as tight as a stereotypical architecture [34] showing how to use every component. In the REBOOT view [24], the architecture shows the connection of abstract classes, with particular concrete subclasses being chosen for a particular implementation using the framework. Choice of components is constrained by the architecture.

Architecture-driven reuse of components originated in a number of European projects (ITHACA [1]; PRACTITIONER [21]) and in the US in the DARPA [11] funded Domain Specific Software Architectures (DSSAs) that evolved within the DoD funded Adage project (e.g. Ref. [36, Appendix A]). These are product-line approaches, where they identify an abstract set of requirements that cover all applications you might reasonably want to build in the domain; one or more architectures that give a solution meeting the requirements of that domain, together with collections of components that might be used within a particular architecture. Product-line approaches appear to be a very productive and promising approach. The stimulating book by Jacobson et al. [23] advocates an architecture driven approach, and this is the approach taken in the IBM SanFrancisco project [22] which deliver application frameworks and javabean components.

This architecture driven reuse suggests that you should be able to generate solutions using a special purpose programming language. These can be set up to be interactive, capturing design decisions as required. This has been called “generative reuse” by its originator, Don Batory [4], and is an approach that can be expected to be developed further.

4. Reuse as a project activity

We now move to consider the process of reuse, and the people involved and the commercial forces that drive its introduction. Way back, in 1968, our forefathers began to worry about how software was developed, and gathered in a NATO workshop to discuss the problems and initiate many of the solutions that have been developed successfully in software engineering. Among the contributions was a paper by Doug McIllroy [25] pointing to the success of libraries of mathematical routines, indicating a future of software components. This early paper really did not influence later developments, but does indicate the thinking of the time: the focus was on the reuse of code, and McIllroy said “mass-produced” components to emphasise the industrial model he had in mind.

During the 1980s, the idea of reuse was broadened by a number of people to include things developed in projects other than code, so that in 1995, Karlsson [24] could write:

The essence of software reuse is to use any information, artefact or product held in the software company’s inventory. The reusable element can be code, requirement or design specification, or knowledge of the domain. It can also include processes, methods and templates which are known to be effective in specific cases. [24, p. ix].
However, today the emphasis has swung back to code, with business objects and the ideas of reuse emerging in “third generation” OO methods like Catalysis [14]. Alas, as emphasised above, the emphasis has been on small components.

The reason for undertaken reuse programmes to Jacobson et al. [23, p. 5] was “faster, better, cheaper”, and was summarised by Karlsson as:

Reuse is not an end in itself, but a means of achieving the general objectives of the company. Companies today are faced with new and more challenging market pressures. In response, companies have to reduce time-to-market with new or enhanced products, increase diversity of products available to customers, and enhance the standardisation and interoperability of the products. Reuse is a means to achieve such objectives. [23, p. 3].

Missing from these is also an oft-repeated quality gain, since a component will, it is reasoned, have been very thoroughly tested. Weyuker [38], among many others, has very properly warned against this view—when reusing a component it will most surely be used differently, and prior testing may not predict reliability in its new context: the fate of the Ariane 5 space rocket is a salutatory warning.

We are led to viewing reuse as happening within a company through the development of a library of reusable assets, and within a project in considering the reuse of assets from the library and perhaps contributing to the library of assets developed by the project. We might draw a lifecycle diagram like in Fig. 2. Domain analysis identifies the components required and the hotspots necessary to allow customisation, and might also develop the domain specific architectures to guide their use. We might also buy in components from commercial companies and incorporate these into our asset repository.

In spite of considerable progress in the technical side of reuse, getting reuse to actually happen has proved difficult. Rewards for reuse may not be gained by the reuser, and there can be considerable human resistance to reusing things. Software engineers are often highly educated and creative and they really like inventing things themselves; they may also be reluctant to trust software from elsewhere, and want to be in control of the forward development of their software. Many people have pointed this out, Tracz [36] brings together many of these concerns in his entertaining book.

Jacobson et al. [23] give a number of success stories from HP and Ericssons and AT&T—up to 95% reuse, time to market reduced 2–5 times, defects reduced 5–10 times, with overall costs including the cost of developing the asset base still being reduced by 15–75%. These are very impressive benefits, only obtained in architecture-driven reuse.

5. An industrial view of reuse

This emphasis on the reuse of code, and more recently on other “assets” produced during software development, has led to bad press for our industry, implying that software developers are irresponsible by not reusing these assets. To my mind, this view of reuse has always missed the point of how our industry does build upon the foundations of former activity.

Wegner [37] described software development in 1984 as a capital intensive industry in which a lot of money was spent at the start, investing in software to run our software (operating systems, and now middleware), and the software tools to help us develop the software more effectively. Now that is more like it, but he went wrong when equating reuse with repeated execution, rather than with redeployment within a new software development project. But he was right, layered software enables a new development to
achieve more by building upon established layers of system software, and I have argued the same [20] for programming languages and compilers. When I made this point in a seminar in Newcastle in the late 1980s, I was shouted down, “compilers are just tools”, but that misunderstands what a tool is, the embodiment of the prior experience of the industry, enabling us to reuse that experience.

In 1986, Brad Cox [9] wrote a book on OO Programming, in which he introduced the idea of “software integrated circuits” which he has pushed persistently ever since as the solution to software development problems. He envisaged an active market buying and selling components. When Brooks [6] expressed his concerns about software development, saying there was no single magical “silver bullet” solution, Cox [10] responded to re-emphasise his view of components as the solution.

The trouble is that this Coxian view of industrial production is out of date, based on Fordist and Taylorist ideas of mass production through specialisation and repetition. Taylorism is implicit in our use of lifecycle models, and our division of our labour into analysts and programmers/coders. For nineteenth century and early twentieth century manufacture this worked fine, but it does not work today where much greater flexibility is required to enable customisation and responsiveness to market pressures. There is a wider concern about software engineering being rooted in a rationalist and modernist view of science and engineering, a concern expressed elsewhere (e.g. Ref. [33]).

Today we think of knowledge management and learning organisations. Information is necessary for successful manufacture, but in information-based methodologies the assumption is that environments are predictable and stable and the emphasis is on data accuracy, integrity and consistency. However, today’s global environment is continuously changeable and unpredictable, and information on its own is not very useful. Information together with an effective and useful interpretation is what constitutes knowledge. Unlike information, the process of knowledge creation can occur only as a result of a process of social interaction between people. Nonaka and Takeuchi [28] argue that the knowledge creation process goes through a cycle of four stages as knowledge moves from being tacit and internal to explicit and external and back again: a process of socialisation converts tacit knowledge between individuals, being developed and enhanced as it goes; a process of externalisation then converts from tacit knowledge to explicit knowledge, possibly through some elicitation method; a process of combination next extends and enriches explicit knowledge by some formal process; and finally a process of internalisation converts the enriched explicit knowledge to tacit knowledge, in training programs for example.

What we observe in companies undergoing reuse programmes is knowledge management, but uninformed by developments in business theory. There has been one exception, an early paper by Guillermo Arango, which I have not been able to locate, which discussed at a theoretical level the relationship between software reuse and learning organisations. Later Arango et al. [2] put this into practice in developing technology books in order to capture the knowledge of designers at Schlumberger before this experience was lost to the company through retirement. What was captured was not components, or even software, but rather knowledge; the Nonaka and Takeuchi externalisation process was carried out.

Design is a pervasive activity across engineering. Theoreticians recognise two varieties of design—creative design and routine design. Most books on design theory focus on creative design, and it was only in 1986 that Brown and Chandresekan [7] explicitly identified routine design in the context of seeking to automate design processes. Design theory has been applied to computing by Dasgupta [12] who characterises routine design as:

...those types of design problems which are well understood or mature in the sense that the form of the artefact, its method of design, and the mode of its manufacture... are known before the design process actually begins... The task of the designer is essentially ... to meet the particular requirements posed by the design problem. (p. 319)

Architecture driven software reuse is routine design. Biggerstaff, quoted by Tracz [36] says:

If you have not built three real systems in a particular domain, you are unlikely to be able to derive the necessary details of the domain required for successful reuse in the domain. (p. 15)

ERP (Enterprise Resource Planning) solutions, such as SAP (e.g. Ref. [3]) are the extreme of routine design and reuse, providing total solutions with little room for customisation to meet the particular requirements. The user must adapt to the software. They claim to be based on business objects and components, but need to develop considerably before they can exploit the flexibility that claim might imply.

Creative design is something else, and Jacobson et al. [23] are mistaken in believing that a novel problem can be routinised to apply architecture driven reuse. Innovation requires less rigid approaches, patterns.

Recently, there has been much activity in design patterns (e.g. Refs. [8,17]): these “record experience in designing object-oriented software” and “make it easier to reuse successful designs and architectures” [17, p. 2]. This idea has been widely taken up in the computing community, so that we now find analysis patterns, business patterns, HCI patterns and even teaching patterns. And so it should be, this is the community viewing itself as a learning organisation, practising the externalisation of tacit knowledge within knowledge management. Further, patterns can be applied in any order, though of course there are dependencies—but there is no rigid lifecycle. Patterns and their use are essentially post-Fordist.
Indeed, we can view all the asset collection activity, particularly the domain analysis and collection of design frameworks and software architectures, as a form of knowledge management. The benefit to the organisation, and to the industry if the knowledge is shared as it is with patterns, is that tacit knowledge has been externalised and made amenable to sharing, to give the organisations and industry the flexibility needed to be able to respond to a rapidly changing world. Arguments focusing on single organisations and projects or the trade in components in isolation just miss the point.

6. Salvation in the next millennium?

Architecture-driven component reuse is the way forward for the bulk of standard routine applications. OO methods need to be developed to accept and use a component model that is more appropriate than the traditional object (class), and to aim to reuse components guided by architecture. The book by Jacobson et al. is an excellent first step but needs to recognise that this approach is for routine design and back off innovative problems. The exemplars of current industrial best practice are those cited by Jacobson et al., and IBM’s San Francisco project.

For innovative problems, do not expect systematic reuse let alone an architecture-driven approach. Exploit knowledge of software development as embedded in tools and available in patterns. Be opportunistic about any other reuse.

References