1. An Introduction and History of Software Architectures, Components, and Reuse

Leonor Barroca, Jon Hall and Patrick Hall

Summary

Software components and architectures are gaining considerable support as the way to develop object-oriented systems and business applications. Current developments in component-based software reuse is taking place in the context of some thirty years of history. It is as well to rehearse that history, as we do in this chapter, and draw lessons from it, lest we repeat the errors of the past.

This chapter fills in the background to the rest of the book. Concepts and ideas assumed in the other chapters will be explained and set in a context which will aid the reading and understanding of those chapters. The specialist reader already conversant with the field may wish to skip this chapter, though it is hoped that even for such a reader the orientation adopted here will be informative and helpful.

1.1 Introduction

Software architectures have become very topical and important during the second half of the nineties. Why now? After all, the ideas go back thirty years to the origins of software engineering — 1968 — with a first cycle of active interest in components and reuse in the real-time area in the early 1980s. Now, in the late 1990s, the resurgence of interest is driven from the business information systems end.

This renewed interest in components has, of course, encountered the same problems that were encountered previously; moreover, similar solutions are appearing. However, not only have we learnt much in the intervening years, but the technology has also evolved, solutions are potentially better, and more likely to endure.

How to do better this time round is the theme of this book. In it, we have chosen the ideas and technologies that give the current interest in software architectures a bright future.

1.2 Software Architecture

Since the inception of software engineering, software architectures have always been with us. Going back to the 1960s, when systems were comparatively small, we all drew abstract diagrams in order to understand designs and communicate them to others. These small systems did not always need abstractions to help in the design process; nevertheless, they helped. However, very soon, systems grew in size and ambition, and structured descriptions became essential, were used very naturally — more

or less without thinking — and derived from similar practices in other branches of engineering. In some military projects several levels of description were mandated, for example in the UK through the military standard JSP188 which was really aimed at hardware. JSP 188 mandated four layers of abstraction and the software design method MASCOT [27] was developed to assist in this.

The highest levels of design are what we now know as software architecture. The practice at the time was to sketch ideas and to pin the resulting (large!) diagrams on the walls of offices. Later on in a project you would find that what was on the wall had a limited relationship to what was actually being done; everybody knew that it was out of date — it was just "inspirational decoration". Moreover, there was no need to update an architectural diagram because everybody in the team understood what was actually being built. In database design there were also levels of abstraction, initially between logical and physical schemas, later with the introduction of conceptual schemas; this gave three, effectively architectural, layers of abstraction.

Although not expressed at the time, even at the very beginnings of software development practice, we used architectural level descriptions, initially just as design aids; later, as the descriptions became essential for maintenance purposes, they were also preserved and maintained.

In 1976, De Remer and Kron [10] published their seminal paper on "programming in the large" versus "programming in the small". This paper gave us the means of talking about systems at two levels, the level of job control language (JCL) and the level of sequential programs that were controlled by the JCL. This led to module interconnection languages (MILs), a fruitful line of research over the past 20 years. Programming in the large was represented in the operational system as JCL or similar, and was the architectural level of description as captured by the MIL. Programming in the small produces modules or components which could, in principle, be reused by putting them together in some new way at the architectural level by programming in the large.

Software architecture did not itself gain currency until the 1980s; perhaps the first book about software that appeared with "architecture" in its title was by Best in 1990 [4]. However the idea of software architecture was then current, as seen in the DARPA workshop in 1990 [9]. Subsequently the seminal book on software architecture has become that of Shaw and Garlan [26], which brought together much of the work of these important pioneers of the subject.

Later chapters of this book pick up and build on this work of software architectures. The essential idea of ***software architecture*** is that software at a high level of abstraction can be described as a number of distinct elements or subsystems together with their interconnection and interactions. This will be elaborated further below.

We will see later in this chapter why architectures are so important, not just as a level of abstraction to enable the design of large systems, but also to enable the effective reuse of software components.

1.3 Reusable Components

In 1968, at the celebrated first ever software engineering conference, it was realised that this stuff called software was really getting very difficult and too large to manage and that we needed different ways of handling it. The genesis of reuse and components began then, with a paper by Doug McIlroy [19] who talked about software components, but just as utility libraries or libraries of mathematical routines, not at all what we would now think of as software components — as useful fragments of a software system that can be assembled with other fragments to form larger pieces or complete applications. Nevertheless his vision has been proved correct.

In the 1970s, structured methods emerged, principally in the US, while data centred design emerged in Europe. In the late 1970's, a project called DRACO, in California, was started by Peter Freeman [21, 12], the first software reuse project of which we are aware. In that project, the ideas of components, levels of abstraction, and domain analysis, were all introduced. The module interconnection languages started by De Remer and Kron [10] were used on DRACO, and developments of these were pulled together in an excellent survey paper by Ruben Prieto-Diaz and Jim Neighbors in 1986 [23].

The next burst of activity concerning software components was in the mid 1980s when people first began to talk about the industrial organisation of software development

In 1984, Peter Wegner [29] described software development 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. Software reuse was an important theme of his paper; he did, however, equate reuse with repeated execution — as when we use facilities of the operating system — rather than in the now current view of redeployment within a new software development project. In 1986 Brad Cox wrote a book on object-oriented programming [13], in which he introduced the idea of "software integrated circuits" (ICs) which he has pushed persistently ever since as the solution to software development problems (e.g. [7]). Software integrated circuits are lumps of software that you can plug and unplug and replace and reconfigure, sell on the open market, and so on. He developed a Smalltalk-like language called Objective-C that could be embedded in C and pre-processed to C, but which lost out in competition with C++.

Cox's software integrated circuits are our software components. Figure 1.1 shows the general idea of components — pictures like this have been drawn by many authors, though this particular style of doing so originated with Peter Welch. The plugs and sockets are the interfaces, subdivided into coherent parts. Plugs show interfaces called (required), while sockets show interfaces offered (provided). Where the shading is the same, the interface parts are of the same type, and plugs can be connected to sockets of the same type.

Note in particular that interfaces are subdivided into coherent parts, and that the set of interfaces required of other components is made explicit.

These kinds of components might equivalently be represented in text using some component description language, something like that shown in Figure 1.2. Note that

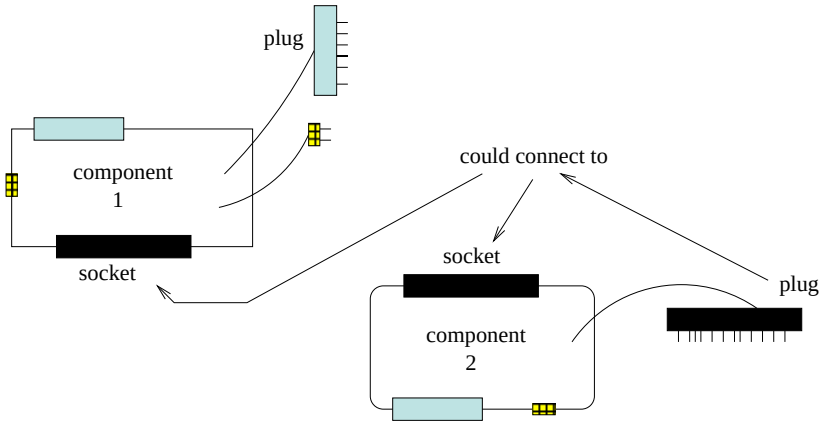

Figure 1.1. Software components

we have also parameterised the component, as one might wish to do, for example, for a generic stack that can handle various types of objects. However, it is not clear that generic parameters are any different from required interfaces, since a parameter is just a class which could have been specified as a required component to be "given a value" using a connection.

```
component Comp1 (param1, param2, param3);
  interface offered Socket1, Socket2;
  interface required plug1, plug2;
  component body
    . . .
end component;
. . .
connections
  join Comp1.Plug2 to Comp3.Socket1;
    . . .
end connections
```

Figure 1.2. Textual definition of components and interconnections

In the mid 1980s, the talk of the software crisis and the skills shortage, made industrialisation approaches very attractive. A lot of government research and development funding went into reuse projects both in the UK in the Alvey programme in the early 80's, and also into various European R&D programmes. One such project was the EU funded ITHACA project [1], which has been credited (by Wirfs-Brock and Johnson [30]) with introducing object-oriented frameworks. Another EU project, PRACTITIONER [15], also talked about frameworks at about the same time, but not in an object-oriented context, taking its inspiration, through Peter Elzer, from architecture and Alexander before this became fashionable.

The last project of the EU programmes to be focused exclusively on reuse was REBOOT [20, 11], which ended in 1994.

During the period of time over which these projects ran, a view of how software reuse fitted into the lifecycle emerged, as shown in Figure 1.3. The reuse process is built round a component library into which components are added after component engineering, and from which components are taken during the development of new software.

The European approach was often characterised by lots of sophisticated theory, particularly on module interconnection languages. Joseph Goguen [14] developed LIL, the Library Interconnection Language (further developed by Tracz [28], based upon theory from Rod Burstall). Cramer and others [8] produced a language called Pi. These approaches defined a language like that shown in Figure 1.2 for both describing components and for describing their interconnection, with this language being given both formal syntax and formal semantics.

There were also many sophisticated approaches about how to organise your components into libraries so that you could find them readily, using search mechanisms based on information retrieval and the experience of the library sciences (e.g. [10, 24]). The basic idea in most approaches was to identify components using keywords, possibly organised hierarchically, and then use indexes to find matching components, with a final round of manual checking to select the component that actually met the requirements. Attempts were made to represent the functionality of the component mathematically, and use this as a basis for retrieval, but led to problems of formal undecidability as there is no general algorithm for matching a search expression with the mathematical specification of a component.

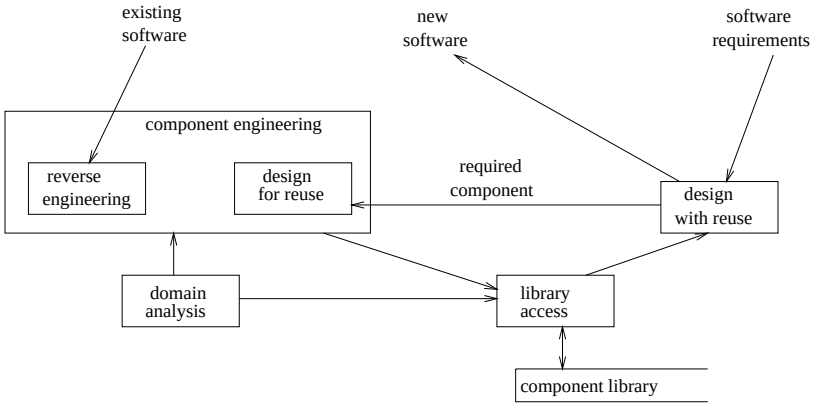

Figure 1.3. The Reuse Process

There had been considerable progress in describing components and their interconnection, and in developing and managing libraries of components. All of this had been focused upon technical processes. It did not work!

1.4 Setting a Context for Component Reuse

This is the story of our technology, is it not? Lots of great ideas — forget the last "solution", we promise the next one will work. Then we all work like mad at it, but fail to deliver the promise. As way of reward, however, we all learn a lot.

We have learnt that reuse works in a narrow domain. In 1987 Grady Booch developed a library of low-level Ada components [5] 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 not achieved, and the project was abandoned [18]. In contrast the HP telecommunications facility in Grenoble managed to use components very effectively for telephone switching systems, but they only had about 100 components. That HP 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.

One lesson about successful reuse strategies was that you had to focus on a particular area of application, called a domain, and collect only components which are definitely known to be useful in that domain.

Another lesson was that collections of components would be small and, therefore, that library retrieval systems were 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.

In recognising the value of focusing on a domain, domain analysis emerged; in choosing components for a narrow domain, like the telecommunications domain of HP in Grenoble, you need a good understanding of the area in which you are working. You need to understand what it is you are trying to build, and then identify components that have been specifically created for that kind of software. But domain analysis is more powerful than that, and at Lucca, in 1993, Guillermo Arango [2] described work with his employers, Schlumberger, as going around the engineers and carefully documenting what was in their heads in "technology notebooks". Although this has little to do with building software systems, quite a few people were inspired by this as a way of reducing dependency on key software people, and the knowledge they have, by externalising tacit knowledge.

Focusing on narrow domains was still a technical measure, but it was also noted that failure came about for other non-technical reasons. There was human resistance to reusing things. Software engineers are often highly educated and creative and enjoy inventing things, so there can be a lot of resistance to picking up others' software and using that instead of having the fun of writing it themselves. They were also reluctant to trust software from elsewhere often feeling that they needed to be in control of the forward development of the software. Moreover, there was no real incentive to develop software for reuse, since work was focused on development projects in isolation. All of these issues need management resolution. At the first European conference on software reuse in 1991 there was a clear division between the European contributors taking a very theoretical approach, hoping that the technology would solve the

problem, and the American and Japanese contributors who talked about management problems.

The managerial measures that can be taken are numerous, and many companies today will have projects for building component libraries, with incentives for contributing to, and using components from, the library.

1.5 Components and How to Use Them

We now have experience that components alone are inadequate; what is needed in addition is to think about how we use the components from the point of view of:

- the domain in which they are deployed, and how to use that high level domain knowledge to guide the application;
- the management processes being used to encourage the use of the components and, thereby, obtain costs and time-scale reductions.

These observations lead to the very simple idea of Domain Specific Software Architectures (DSSAs) that evolved within the DoD funded Adage project [9] in the United States. The idea is to identify an abstract set of requirements that cover all applications you might reasonably want to build in the domain. You have a collection of architectural descriptions to explain how you might solve problems in the domain, together with collections of components that might be used within a particular architecture. Hewlett Packard have been rumoured to be representing their product lines as DSSAs. As early as 1986 a product-line for CAD software had been considered [3]. Product-line approaches appear to be very productive and promising; we present possible product-line descriptions later in two chapters of this book.

These domain architecture driven approaches suggest that you should be able to generate solutions in an approach like 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 [3], and is an approach that can be expected to be developed further.

A key requirement in domain specific approaches is the need for stereotypical architectures with slots for components. These are now known as frameworks, usually associated with object oriented approaches, though the concepts are much broader. Indeed the usual view of objects which does not include explicit required interfaces needs extension in this direction.

1.6 Current and Future Developments

Since the early 1990s, and the seminal book [8] by the “Gang of Four”, Gamma, Helm, Johnson and Vlissides, patterns have achieved much prominence. Patterns can be viewed as reusable experience, for they catalogue commonly recurring problems, within a context, that designers and analysts have encountered and give one or

more solutions that are known to work well. In addition, other information might be recorded in a pattern, one example being the consequences that flow from the choice of a particular solution.

Since this original work on design patterns, the idea of patterns has been applied to many areas, including human computer interaction, education, and business. What patterns do is capture problem solving experience, making explicit the tacit knowledge of the experienced analyst or designer. We took this same view of frameworks earlier in this chapter; indeed, many other researchers do not distinguish between patterns and frameworks. However, there is a principled distinction to be made: frameworks record solutions for an application in a specific domain, while patterns record solutions for a particular smaller problem not necessarily in a specific domain. Under this interpretation, patterns can have increased significance in the development of new frameworks, as well as in their documentation.

The significance of the emergence of patterns should not be underestimated. Patterns interact, and are applied in the order that the experience of the user of the patterns would indicate. They appear to have gained extra popularity as a reaction against the highly prescriptive structured methods like Yourdon and SSADM; just as the heavily technical early approaches to components failed, so the heavily mechanist approaches to software development methodologies also failed.

In the late 1990s business components have gained prominence. This seems to have happened as a result of object-oriented approaches slowly gaining acceptance. Objects have been acclaimed as the software component technology ever since Brad Cox's software ICs. They appear in Corba, which has since emerged as one standard way to distribute applications.

However, the current developments in object-oriented approaches, and the re-emergence of components as business components, stands in real danger of overlooking the hard-won experience gained in the real-time area where Booch and others began.

From the account, earlier in this chapter, of how things developed in the real-time arena, you might expect architectural level descriptions to emerge in the new fields of application, and, indeed, they are, as "application architectures" and "industry models". The San Francisco project [25] is just one of many such initiatives, in this case based on Java components. The success of such industry-model driven approaches is already claimed, though objective assessment is still required.

This book continues the development of component-based approaches to software development.

Underpinning the work on software architectures, we need a sound model of a component, grounded in theory but with an understanding of the use of components in constructing applications. Chapter 2 is one possible approach to this, separating out, as it does, components from the way they are connected or composed, adding the concept of "glue", code which bridges any mismatches between the interfaces of components.

One hard-learnt lesson is that objects are not enough. To achieve distribution, decomposition into a number of collaborating objects is needed, and these together

should be viewed as the component. Individual objects are distributed. Chapter 3 gives an excellent account of how this is achieved, and the way the objects within a component are distributed between client and server, from the point of view of a single component.

Chapter 4 presents an object-oriented method for designing components and kits of components. The approach described, part of the *Catalysis* method for developing object-oriented software, is one of the most comprehensive and best available. The approach produces coherent kits of components without, however, going as far as architectures.

Chapter 5 argues that objects and components require architectures, since it is architectures, and particularly architectural styles, that determine the non-functional properties of a design. This, in turn, determines how functions are distributed in the system, as illustrated by a very convincing example.

Accepting that architectures are important, their systematic design becomes an important issue. Chapter 6 gives us one method for doing this, following an iterative approach of evaluating a candidate architecture, followed by transforming it to improve the shortcomings identified in the evaluation. In Chapter 7 we see an example architecture for a system distributed on the Internet.

Earlier in this chapter we saw that it was not only single architectures that were important, but that architectures were the embodiment of a range of related applications which shared an architecture but differed in the specific components used in that architecture. This theme is picked up in two chapters on product-lines. In Chapter 8 we see how parts of an architecture, where customisation is expected, can be focused on through “framelets”. In Chapter 9 we see the practical use of product-lines in industry illustrated with two case studies.

Software development takes place within the context of an enterprise (or enterprises) that will benefit from that development, and a range of technologies available for use in that development. Chapter 10 gives an overview of these technologies and their importance to business.

In this chapter we have reviewed the state of development of component and architecture based software development, setting this in the context of software reuse research from 1968 onwards. In Chapter 11 a review of the current state of development is presented from the context of experience with user interface systems. The arguments of that chapter are complimentary to this chapter; the conclusion is, however, the same — that component-based software development must learn from past experience in the area, lest it repeat past failures. Finally, in Chapter12 we hear what is happening in industry where the uptake of components and architectures has been going on for a while, and where the main lessons are being learnt.

References

1. Ader M, Nierstrasz O, McMahon S, Mueller G, Proefrock A-K. The ITHACA Technology: A Landscape for Object-Oriented Application Development. In: ESPRIT 90 Conference Proceedings. European Commission, 1990

2. Arango G, Shoen E, Pettengill R. Design as Evolution and Reuse. Advances in Software Reuse. Selected Papers from the Second International Workshop on Software Reusability, March, 1993, Lucca, Italy, IEEE Computer Society Press, pp 9–18
3. Batory D, O'Malley S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology 1992;Vol. 1, No. 4:355–398
4. Best L. Application Architecture: Modern Large-Scale Information Processing, John Wiley & Sons, 1990
5. Booch G. Software Components with Ada, Structures, Tools and Subsystems, Benjamin/ Cummings Publishing Company, Menlo Park, 1987
6. Cox B. Object-Oriented Programming — An Evolutionary Approach, Addison-Wesley, Reading, 1986
7. Cox B. There is a Silver Bullet. Byte 1990;Vol. 15, No. 10:209–218
8. Cramer J, Fey W, Michael G, Grosse-Rhode M. Towards a Formally Based Component Description Language — a Foundation for Reuse. Structured Programming 1991;Vol. 12, No. 2:91–110
9. DARPA Proceedings of the Workshop on Domain-Specific Software Architectures, 1990. Available from the DSSA Program Manager. DARPA/ISTO, Arlington, VA 22209, 1400 Wilson Blvd
10. De Remer F, Kron H. Programming in the Large Versus Programming in the Small. In: IEEE Transactions on Software Engineering, June 1976, pp 312–327
11. Frakes W, Nejmeh B. Software Reuse Through Information Retrieval. SIGIR Forum 1986/87;Vol. 21:1–2
12. Freeman P. Conceptual Analysis of the Draco Approach to Constructing Software Systems. IEEE Transactions on Software Engineering, 1987 and included in IEEE Tutorial: Software Reusability, 1987
13. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns — Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995
14. Goguen J. Reusing and Interconnecting Software Components. IEEE Computer February 1986;Vol. 19, No. 2 (Feb):16–28
15. Hall P, Boldyreff C, Elzer P, Keilmann J, Olsen L, Witt J. PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software, Ancillary papers at ESPRIT week, November 1990
16. Hall P. Software Components Reuse — Getting More out of your Code. In: Tracz W (ed) Information and Software Technology Butterworths, January/February 1987. Reprinted in IEEE Tutorial, Software Reuse: Emerging Technology, IEEE Computer Society, 1988
17. Karlsson E-A. Software Reuse. A Holistic Approach, John Wiley & Sons, New York, 1995
18. Kruzela I, Brorsson M. Human Aspects and Organizational Issues of Software Reuse. In: Hall P (ed) Software Reuse and Reverse Engineering in Practice, London, Chapman & Hall, 1992, pp 521–534
19. McIllroy M. Mass-Produced Software Components in Software Engineering Concepts and Techniques, Petrocelli/Charter, Belgium, 1976, pp 88–98
20. Morel J, Faget J. The REBOOT Environment. In: Advances in Software Reuse. Selected Papers from the Second International Workshop on Software Reusability, March, 1993, Lucca, Italy, IEEE Computer Society Press, pp 80–88
21. Neighbors J. The DRACO Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Engineering 1984;Vol 10, No. 5(Sept)
22. Prieto-Diaz R. Domain Analysis: an Introduction. ACM SIGSOFT Software Engineering Notes 1990;Vol. 15, No. 2 (Apr):47–54
23. Prieto-Diaz R, James N. Module Interconnection Languages. Journal of System Sciences 1986;Vol. 6, No. 4 (Nov):307–334
24. Prieto-Diaz R. Implementing Faceted Classification for Software Reuse. Communications of the ACM 1991;Vol. 34, No. 5 (May):88–97

25. See http://www.software.ibm.com/ad/sanfrancisco/
26. Shaw M, Garlan D. Software Architecture — Perspectives on an Emerging Discipline, Prentice Hall, Upper Saddle River, New Jersey, 1996
27. Simpson H. The MASCOT Method. Software Engineering Journal 1986:103–120
28. Tracz W. LILEANNA: a Parameterized Programming Language. In: Advances in Software Reuse. Selected Papers from the Second International Workshop on Software Reusability, March 24–26, 1993, Lucca, Italy, IEEE Computer Society Press, pp 66–70
29. Wegner P. Capital-Intensive Software Technology. IEEE Software, July 1984
30. Wirfs-Brock R, Ralph E. Surveying Current Research into Object-Oriented Design. Communications of the ACM 1990;Vol. 33, No. 9 (Sep):104–124