Automatic component substitution for distributed component-based systems

Michael Alcock

Master of Philosophy

September 2006

Keele University

Abstract

Software components are reusable sets of functionality that can be created, deployed, discovered and integrated into applications by third parties. They can be used together and reused in different combinations to perform various tasks. Such software reuse is expected to reduce time-to-market and cost by avoiding the need to 'reinvent the wheel'. Distributed environments exist in which *integrators* request components to use and *suppliers* provide components or their functionality.

Distributed component-based systems suffer from the same perils that using networks bring. To preserve the operation of a system in the event that a component ceases to be available, an appropriate replacement component must be found. To minimise system downtime during this process, a quick, automatic way of comparing, selecting and combining components whose functionality matches those being replaced is desired. Components' purpose, functionality and usage details should be described. This description can then be used in *Automatic Component Substitution (ACS)*, in which replacement components are described and searched for automatically and selected for integration.

Where an identical substitute component cannot be found, components capable of supplying needed functionality can be searched for regardless of whether they are intended or advertised for that purpose. This concept promises greater success in selecting appropriate alternative components. It is described as *near-match ACS* (NMACS). Exact- and near-match ACS are described collectively as (NM)ACS.

This document explains and analyses the requirements for (NM)ACS and some of those component description languages currently available to support ACS. Three candidate technologies are evaluated and shown to ably represent and support these requirements. A component description and set of description properties is also proposed. A case study is then evaluated using the Goal Question Metrics evaluation method against the suggested description and the requirements. This exemplifies that substitution can successfully be implemented for both exact- and near-match components.

Table of Contents

1 Introduction 8
1.1 What This Work *Will* Concentrate On 11
1.2 What This Work *Will Not* Concentrate On 11
1.3 Assumptions Made in Suggesting a Component Selection Strategy 12
1.4 Assumptions Made in Evaluating the Ability of Web Technologies to Facilitate a Component Selection Strategy 13
1.5 Limitations of this Work 13
1.6 Criteria for Success 15
1.7 Aims 16
2 Literature Survey 17
2.1 Reuse 17
2.1.1 Types of Reuse: An Overview 21
2.1.2 Adaptive Reuse 23
2.1.3 The Problems of Reuse 25
2.2 Distributed Systems 26
2.3 Repositories 28
2.4 Components 29
2.4.1 Furthering the Component Definition 31
2.4.2 Overall Component Definition 35
2.5 Component Descriptions 36
2.5.1 Keyword-Based Descriptions 38
2.5.2 Structural Matching Descriptions 42
2.5.3 Fuzzy Matching and Classification 44
2.5.4 Non-Functional Descriptions 48
2.5.5 Precision and Recall 53
2.5.6 Limitations 54
2.6 Some of the Emerging Fields of Distributing Systems 55
2.6.1 Grid Computing 55
2.6.2 Web Services 57
2.6.3 Software as a Service 60
2.6.4 Components Off-the-Shelf 61
2.6.5 CLARiFi 62
3 Component Substitution Descriptions 65
3.1 The Issues of Precision and Complexity within Automation 67
3.2 Exact- and Near-Match Components 68
3.3 Component Domains 69
3.4 Property Data Types 70

3.5 Property Requirements 71

4 Component Substitution Strategies 74

4.1 The Matching Process 74

4.2 Ways of Implementing the Description Language 75

4.2.1 The Property Value Gathering Process 75

4.2.2 A Suggested Selection Process 76

5 Evaluation Strategies 82

5.1 Choosing an Evaluation Method 82

5.2 Explanation of Feature Analyses and Approaches 82

5.3 Choosing a Feature Analysis Approach 84

5.4 What to Evaluate 85

6 Evaluation of Web Technologies 89

6.1 The Candidates 89

6.1.1 The Enterprise JavaBeans™ Specification v2.0 90

6.1.2 Web Services Description Language (WSDL) Specification v1.2 99

6.1.3 Distributed Component Object Model (DCOM) v1.0 102

6.2 Scoring System 108

6.3 The Scoring 108

6.3.1 The candidate should facilitate the NMACS properties 109

6.3.2 Property values should be generated and written automatically 110

6.3.3 Property values should be ably written as and when needed 111

6.3.4 Property values should be ably accessed as and when needed 111

6.3.5 Components should be substituted as and when needed 112

6.3.6 Component suppliers should be ably switched as and when needed 112

6.3.7 Total Scores 113

7 Evaluation of (NM)ACS – Case Study 114

7.1 Overview 114

7.2 The Swing Components Used in the Wizard GUI 116

7.3 Overview of Wizard GUI's Structure 117

7.3.1 The Design Rules 118

7.3.2 Operation of the GUI Wizard 120

7.3.3 Error Messages 121

7.4 The GUI Wizard's Appearance 122

7.4.1 The Appearance of Error Messages 122

7.4.2 Page 1: Ask Whether the User is Registered 122

7.4.3 Page 2: Ask for Departure, Destination and Whether it is a Return Trip 123

7.4.4 Page 3: Ask to Select a Ticket Type 123

7.4.5 Page 4: Show Ticket Summary and Price 124

7.5 Technical Details concerning GUI Construction 124

7.5.1 Constructing Pages 125

7.5.2 Page States and User Data Validation ... 127
7.5.3 Storage of System State ... 127
7.5.4 Deciding Page States ... 129
7.5.5 Middleware Functions ... 129
7.5.6 The Components in the Repository ... 130
7.5.7 The Structure of the Repository ... 132
7.5.8 (NM)ACS in Use ... 133
7.6 Interpreting the Results - Overview ... 137
7.7 Evaluation of the (NM)ACS Approach ... 137
7.7.1 Questions ... 138
7.7.2 Metrics ... 139
7.8 Limitations ... 140
7.9 Improvements for the Future ... 141
8 Discussion ... 143
9 Conclusion ... 144
10 Appendices ... 146
Appendix A List of Components in the Repository ... 146
Appendix B Source Code Listing of Components in the Repository ... 151
Component 1: IsUserNameRequired ... 151
Component 2: IsPasswordRequired ... 151
Component 3: ValidateUser ... 152
Component 4: GetStations ... 153
Component 5: ValidateDepartureAndDestination ... 153
Component 6: IsEconomyClassAvailable ... 154
Component 7: IsBusinessClassAvailable ... 155
Component 8: IsTicketSelected ... 156
Component 9: GetTicketPrice ... 156
Component 10: GetTicketPriceInDollars ... 158
Component 11: GetTicketPriceInPounds ... 159
Component 12: IsBusinessClassOptionHidden ... 161
Component 13: IsUserNameRequired2 ... 161
Component 14: ValidateUserDetails ... 162
11 References ... 163

List of Figures

Figure 1 Types of Software Development Knowledge (modified from Mili et al (1995)) ... 18
Figure 2 Decision Tree for Component Reuse. Source: Ravichandran et al (2003)......... 22
Figure 3 White-box & Black-box Reuse against Costs 23
Figure 4 Stages of Communication between Component Supplier and User.................. 35
Figure 5 Example Faceted Keywords & Weights.. 45
Figure 6 Contract Relationship between Client and Provider 47
Figure 7 Example Pairs of Test Data for Static Behaviour Sampling, in Tabular Form 53
Figure 8 Comparison of Precision and Recall of Retrieval Methods (modified from Bernstein (2002)).. 53
Figure 9 The Roles that Services Play in the Web Services Infrastructure...................... 59
Figure 10 A Proposed SaaS Service Model (taken from Turner et al (2003)) 60
Figure 11 The CLARiFi Component Relationship.. 62
Figure 12 The Details Inherent to a CLARiFi Property 62
Figure 13 The CLARiFi Property-Component Relationship... 63
Figure 14 Example CLARiFi Properties... 64
Figure 15 Pseudocode Representation of Quick Exact-Match Selection 78
Figure 16 Diagram Showing Quick Exact-Match Selection 78
Figure 17 Diagram Showing ‘Dynamic’ Exact-Match Selection...................................... 79
Figure 18 Pseudocode Representation of ‘Dynamic’ Exact-Match Selection................. 79
Figure 19 Pseudocode Representation of Near-Match Selection................................... 80
Figure 20 Diagram Showing Near-Match Selection.. 81
Figure 21 Distinction between Three Java Distributed Architectures 91
Figure 22 A Simplification of the Relationship of, and Information Passed Between, EJB Roles 92
Figure 23 Relationship between Levels of Abstraction in WSDL.................................... 99
Figure 24 Exemplification of the Use of DCOM Pointers (modified from [OPEN99])..... 104
Figure 25 Standard DCOM “Security by Configuration” Architecture 107
Figure 26 Diagram to show flow of information through the case study 115
Figure 27 Diagram to show flow of information through the case study 116
Figure 28 Train Stations and Ticket Prices.. 120
Figure 29 An example of an error message. 122
Figure 30 Page 1, The “User Details” Screen.. 122
Figure 31 Page 2, the “Departure and Destination” Screen 123
Figure 32 Page 3, the “Ticket Type” Screen.. 123
Figure 33 Page 4, the “Ticket Summary and Price” Screen 124
Figure 34 Diagram showing the Relationship Between Parts of the Wizard System 125

Appendices

Appendix A List of Components in the Repository 146
Appendix B Source Code Listing of Components in the Repository 151

Acknowledgements

I would first like to thank my supervisors, Prof. Pearl Brereton and Prof. David Budgen for being my supervisors during the course of this work, and Prof. Brereton in particular for reading through numerous draft copies of chapters and making suggestions on both structure and content. The mass of notes she made during this process was as candid and useful in their detail as they were disconcerting in their volume!

Many Computer Science staff and students completely unrelated to the area in which I have studied were also helpful for numerous reasons, including humour, avoiding work and technical knowledge. These include in particular Phyll Hartshorn, Rashid Jayousi, Shantha Jayalal, Keri Harthorne, Steve Owen, Mark Turner, James Cole, Phil Woodall, Nick Gibbons and all at the Computer Society. The rest of the Department deserves thanks for helping me during my half decade here, who enabled me to earn enough money to stay and complete the work and who provided support when it was needed.

Thanks also to a geographical nightmare of friends, flatmates and associates who have provided mental stimulation and friendly support over the years. These include Gareth Eason, Yijia Lai, Emmanuel Mabunde, Jon Marshall, Xiaoyen Wu and Weiming Zhou (flatmates), Yvonne Benz and Carolin Jöerke (Leipzig), Martin Brown (Jersey), James Dushane and Frances Maxwell (London), Graham Felce (Bristol), Fotis Giakoumakis (Athens), Yukko and Akiko Kurate (Japan), Surjit Nahal (Leeds) and Richard Stead (Liverpool).

Off campus, my sister and parents have been pivotal in forcing me to take breaks from the task at hand by arranging a host of excursions, sports and other activities.

Various musical geniuses have been contributory towards my keeping a clear and innovative mind throughout the write-up stage, who include in my head a nauseatingly random cast of Beethoven, Chopin, Kula:Shaker, Prodigy, Prokofiev, Rachmaninov and Justin Timberlake.

Finally, a special thank-you goes to the Solaris OE for providing brief respite from the tortures of Windows, not least of which was trying to coax Microsoft Word into formatting the text of this document in the desired way.

1 Introduction

As new programming ideologies were introduced, so programming languages moved on to accommodate them. Perhaps one of the most important of these ideologies was to step away from the lineal “if-then-else” way of representing applications as self-contained sequences of instructions, and move towards seeing them as parts of a whole system – objects with attributes that connect to and interact with other objects. This concept is what is now known as *object orientation* (OO), giving rise to a number of languages including SmallTalk, C++ and Java.

The advantages of OO are numerous. OO can make it easier to represent the functionality and execution of a software system and to reuse and extend software objects by plugging them into different systems, thereby cutting out the need to rewrite and repeat sections of code. It enables software models to be applied in a more ‘natural’ way – that is, in terms of objects and relationships rather than as sequences of procedures. In recent years, these ‘objects’ – self-contained units of code that can be ‘plugged in’ with other self-contained units of code – have become known as *components*. With the elaboration of the OO concept to cater for component architecture design and reuse, the school of *component-based software engineering* (CBSE) was born.

From these well-documented beginnings of component-based design have sprung schools of thought related to specific OO contexts. In the past, software objects have been physically close together, often on one computer. Early (and some recent) OO operating systems, which used reusable software libraries, had their resources stored on local storage devices. With the introduction of networking technologies, the scale of potential storage has increased, and software need not be stored on the same computer in order for it to be connected with other software for use in a system. Providing that it is known how to access these software components, they can be used remotely, in a distributed environment, using existing networking mechanisms.

With the growing popularity and success of distributed systems and the improvements in hardware and software design, the capabilities of component-based software engineering can be put to use in new ways. Potentially, if components can be accessed via smaller networks, the same principles can be applied and scaled up in larger, geographically distributed systems such as the Internet. The use of components in distributed environments falls within the school of *distributed component-based software engineering* (DCBSE).

In a locally networked system, the location and number of connected computers is relatively easily identified and managed. This means that components within the network can be located, accessed and utilised easily. However, one problem is that, in much larger, more flexible distributed systems such as the Internet, the location and number of computers is unknown, as computers can be connected or disconnected anywhere and at any time. Furthermore, the potential of large distributed networks also scales up the issues of network speed, security and reliability during inter-computer communication.

In reply to these concerns, and prompted by the increased commercial interest in DCBSE, considerable work has been undertaken to utilise large networks such as the Internet and the Web in a component-based way. Numerous technologies that cater for component registration, discovery, selection and integration and system maintenance and construction exist. However, all are still at early stages of evolution that only cater for basic needs relative to the potential on offer. They have only just begun to satisfy ever-increasing consumer demands. However, the technologies that exist do provide a foundational framework upon which more complex ones can be built. In the same way that successive programming languages build on programming concepts, so distributed protocols and technologies can be built to satisfy the new concepts that are brought up in reply to the possibilities of this vast potential resource.

Distributed component-based systems suffer from the same perils that using networks bring. To preserve the operation of a system in the event that a component ceases to be available, an appropriate replacement component must be found. To minimise system downtime during this process, a quick, automatic way of comparing, selecting and combining components whose functionality matches those being replaced is desired. Components' purpose, functionality and usage details should be described. This information can then be used in the method here termed *Automatic Component Substitution* (ACS), in which appropriate replacement components are described and searched for entirely automatically. The chief benefit of ACS is therefore the lack of a need for human intervention in this process, and possibly the lack of a need for humans to understand how components are described and searched for.

Where an identical substitute component cannot be found, components capable of supplying needed functionality can be searched for regardless of whether they are intended or advertised for that purpose. This concept promises greater success in selecting appropriate alternative components. It is described here as *near-match ACS* (NMACS). Exact- and near-match ACS are described collectively as (NM)ACS.

This document outlines some of the ways in which a description language might be used to carry out (NM)ACS. It then explains and analyses the information necessary so that NMACS can be carried out and suggests some of those component description languages currently available that do, or potentially can, support ACS. Three candidate technologies are evaluated in their ability to represent and support this issue appropriately and thus to support (NM)ACS. A component description and a set of descriptive properties are thus proposed that address the issue of automation. A case study is then evaluated using the Goal Question Metrics evaluation method against the suggested description and the requirements. This exemplifies that substitution can successfully be implemented for both exact- and near-match components.

This work operates within the constraints set out in sections 1.1 to 1.5. Section 1.6 sets out the general criteria that this work intends to show.

1.1 What This Work *Will* Concentrate On

1. Component evaluation and selection
2. The methodologies in which components are described (although it is assumed that component-based systems are capable of supporting component descriptions) (e.g. (Mittemeir, 2002))
3. Component descriptions themselves

1.2 What This Work *Will Not* Concentrate On

1. Component retrieval
2. Repository descriptions and architectures
3. Whether component descriptions are supplied with the component or from a separate source (see (Zaremski, 1995))
4. Component domains (it is expected that the appropriate component domains are decided upon correctly beforehand)
5. The issues of trust (of the vendor), component availability (exclusive group memberships etc) and component quality (of how well the component carries out its functionality). It will be considered enough that a component exists, is available and is capable of carrying out necessary functionality
6. Programming architecture, language and platform dependency issues

7. Abstract data types and abstract data type handling

8. Hardware restrictions, such as limited memory or bandwidth

1.3 Assumptions Made in Suggesting a Component Selection Strategy

1. The component's actual functionality matches what its description (if any) claims it to be

2. There is one version of a component, or otherwise it does not matter which version is used as long as the desired functionality is available (see (Mittermeir, 2002))

3. Components evaluated for suitability can be combined successfully

4. Information about a component's usage necessary for selection is fully available

5. Any negotiating issues, such as licensing and additional resource needs, are known about and resolved beforehand

6. The component to be substituted performed its task correctly and successfully until it became unavailable for use. This ensures that the information gathered from the component to substitute, about what functionality is needed, is sufficiently precise to find an appropriate replacement component

7. The underlying architecture preserves the state of the distributed system such that it can continue normally when a substitute component is found

8. A distributed system is already fully implemented and ready for use, prior to applying (NM)ACS

9. The target distributed system can be globally distributed, contain/use any number of components and be implemented on any number of related networks and computers in a network

10. Appropriate candidate components are available when needed for substitution

11. At least one component can be found that is capable of carrying out the necessary functionality. The expectation here is that the software system implementing the (NM)ACS strategy is suspended until a suitable component is found

12. A given candidate component for substitution is freely available for use, for example isolated testing of functionality

13. Any special needs of a component are catered for prior to the component selection process

1.4 Assumptions Made in Evaluating the Ability of Web Technologies to Facilitate a Component Selection Strategy

1. Each architecture is capable of supporting the underlying software system's underlying framework (e.g. directory structure, inter-component communications)

2. Any architectural dependencies can be resolved outside the suggested selection strategy, in a way that does not impact on the (NM)ACS selection process

1.5 Limitations of this Work

1. There is a dependency on component availability. Time-critical systems may thus suffer if appropriate components cannot be found. However, this is a worst-case scenario. (NM)ACS may be suitable for time-important distributed systems, as the minimisation of human error is an inherent advantage of the strategy

2. Components with insufficient or unavailable descriptions would make (NM)ACS less reliable or impossible

3. Software systems with certain dependencies, such as programming languages and platforms, would not be able to employ the use of (NM)ACS in its current form

4. In situations where components send and receive results composed of abstract (customised) data types, (NM)ACS has no current ability to decompose them into their constituent primitive data types in order to evaluate compatibility. For example, a component may return the correct collection of primitive data types, but they may be enclosed within an abstract data type that is considered incompatible

5. Care should be taken to ensure that the available component vendors can be trusted to provide non-malicious components that are described appropriately to their functionality

6. Architecture-specific components may be much speedier, since they can be optimised for a particular operating system or processor, for example. These advantages have been abandoned in favour of an architecture-independent approach

7. The ideas within this work are not suitable for safety-critical systems in general unless the facilitating Web technology contains features to support them adequately

8. There is little scope for bias towards particular versions of components. As this work stands, any component version is deemed equally suitable if capable of performing the necessary functionality

9. No distinction is made between components on the grounds of available hardware resources (for example, memory, CPU and bandwidth usage)

10. (NM)ACS is not suitable for underlying software that is at the development stage. In other words, the underlying software system should be fully working and tested before (NM)ACS is applied to it. Changing or expanding the system later may require that (NM)ACS functionality be removed and re-applied after the changes have been made

11. Support is not given in this work for components with:

    - Different ordering of signature parameters that are otherwise compatible
    - Differing signature parameter types that can be cast, for example from 32-bit integers to 64-bit integers
    - Components where parameters can be specified in ranges (such as integers between 1 and 10, or enumerations)

1.6 Criteria for Success

Given the constraints, assumptions and limitations described above, the success of the (NM)ACS approach will be deemed to depend upon:

1 Automation of component selection

2 Components being selected that are exact matches to the component previously being used

3 Components being selected that are inexact but near matches, where exact matches are not available

4 Components being replaced automatically by other components

5 Web technologies already exist that are capable of facilitating such a strategy

6 The identification of an example component selection strategy

7 The ability to perform component substitution using only software technologies that are currently available

8 The ability to demonstrate exact- and near-match component substitution in action in a sample scenario

1.7 Aims

Given the criteria for success described above, this work aims:

1 To show that candidate components can be selected automatically

2 To show that components can be selected by matching the exact functionality needed

3 To show that components can be selected by finding an appropriate near match, if no exact match is available, under a variety of conditions such as different parameter order and identical parameters but different output types and values

4 To show that components can automatically be replaced by other components

5 To evaluate a set of Web technologies in their ability to facilitate such a strategy, and to show that one or more of these Web technologies is capable of facilitating it

6 To propose a selection strategy exemplifying how such a strategy might be implemented

7 To show that such a strategy can be used with existing software technologies

8 To show such a strategy in use, using a number of case studies

2 Literature Survey

Reuse is an important focus of distributed systems research, and various reuse strategies have arisen to improve the ways in which they are used. Reuse strategies are employed in the composition and use of distributed systems, which commonly incorporate the use of *repositories.* Repositories store reusable objects. These objects will be referred to as *components*. Components themselves may be structured and represented in a variety of ways, and there are many ways in which they can be described. Each method has its benefits and disadvantages.

The areas of reuse, distributed component-based systems, reusing via repositories, component structures and component descriptions will thus in turn be covered in the remainder of this chapter.

2.1 Reuse

Reuse is widely regarded as being an important characteristic of the quality and usefulness of distributed, component-based software engineering[1] (DCBSE) (Bennett et al 2001; Foster et al, 2001; Frakes et al, 2001; Rine et al, 1999; Szyperski, 1998). A component that can be used in many systems can reduce the need to develop components from scratch[2] (Bouchachia et al, 2001). As pointed out by

[1] For an introductory overview of the debate over whether to reuse, see the Software Development Times article at http://www.sdtimes.com/news/052/special1.htm

[2] A number of experimental technologies (and associated acronyms) aim to facilitate many different reuse methods. These include ConSIT (Fox et al, 2003), MACE and SETHEO (Schumann et al, 1997), NORA and HAMMR (Fischer et al, 1997), TELOS (Chen, 1993) and TERRA (Kwon et al, 1997).

Sutherland et al (2002), the building of components to cope with reuse is an important focus "to meet integration and adaptability requirements and accomplish more effective reuse".

Mili et al (1995) show that reuse can be a useful tool to reduce design complexity. In addition to components, entities that can be reused include software architectures, logical structures, code fragments and higher-level aspects such as working environments and practices (Szyperski, 1998). It involves setting out requirements, finding a component that matches these requirements, and modifying the component if necessary before using it in the distributed system.

Before reuse can commence, knowledge has to be gained about what can be reused. Mili et al (1995) place this knowledge into four broad areas, related to the suppliers and users (customers or otherwise) of these reusable aspects[3] (see Figure 1).

Level	Supplier-related knowledge	User-related knowledge
Environmental	*Technology transfer knowledge* (potential impact, personnel training)	*Utilisation knowledge* (nature of business)
External	*Development knowledge* (project planning and management, cost estimation)	*Application-area knowledge* (underlying application models)

Figure 1 *Types of Software Development Knowledge (modified from Mili et al (1995))*

One problem that reuse has faced is an apparent paradox: If there are no or few components to reuse, or component reuse is difficult, then component users can not justify reuse and thus will not invest in it.

[3] The focus here is on *application-area knowledge*.

Conversely, if there is no investment, little commercial exploitation is likely to ensue in component reuse and there will thus be no components to reuse (Ye et al, 2002). To break this deadlock, it is necessary to enhance the quality, productivity and usefulness of reuse in order to revive an important aspect of component-based systems that has the potential to decline out of use unnecessarily.

In addition to these issues, many third parties may be involved in composing, supplying and integrating components. The reuse of components thus typically involves the following stages in distributed system development (Damiani et al, 1996; Liao et al, 1999; Mittermeir et al, 2002; Szyperski, 1998; Ye et al, 2002):

1. Detail the system specifications (i.e. preferences) and the requirements for functionality
2. Search for components satisfying the requirements from the source(s), e.g. from a library of predefined items, or *repository*
3. Select one or more satisfactory components from the candidate list
4. a) Retrieve the component(s) using a retrieval mechanism

   b) Negotiate the use of the necessary component functionality from the source
5. a) Combine the components in the system, given any necessary adaptations and modifications

   b) Utilise the component functionality in the system without retrieving the component(s)

Many parties may be involved in composing, supplying and integrating components. Thus, in addition to being self-contained blocks of code, components must have associated details that describe how the component can be used and, if necessary, modified (Basset, 1997). This is widely known as a *description language*. Additionally, note that components may (4a and 5a) or may not (4b and 5b) be

retrieved from the point of origin. In the latter case, component functionality may be requested and the results returned as and when needed − i.e. spontaneously, at any stage during the running of the system − from the source.

Much work has gone into software reuse practices. For example, the recent ISO 12207 standard definitions range the whole reuse paradigm. These include (Tilley et al, 2003):

- **Requirements analysis**: the process of studying user needs to arrive at a system, hardware or software requirements definition
- **Architectural design**: the process of defining components and interfaces in order to create frameworks (software systems, or operational environments)
- **Integration**: the process of combining software components, hardware components or both into the framework
- **Qualification**: the process of testing a component's compatibility with the other components in the framework, and with the framework itself
- **Installation**: the process of integrating the component into the framework

Before we continue, it is perhaps best to point out the difference between *browsing* and *retrieval*. As Fischer (2000) puts it, browsing is the process of navigating a library, usually manually (i.e. by human rather than by automated process), in order to find one or more relevant components. Retrieval is the process of finding components and matching them to the requirements (Mili et al, 1995). It should be assumed that the remainder of this document concentrates on the retrieval aspect as opposed to browsing, as a discussion of the structure of repositories, as will be seen, is not a focus of this document, beyond the brief introduction given in chapter 2.3.

2.1.1 Types of Reuse: An Overview

Various authors summarise the two main types of component reuse: white-box and black-box (Bouchachia et al, 2001; Di Felice et al, 2002; Mili et al, 1995; Ravi, 2003; Szyperski, 1998).

White-box reuse includes situations where reusable components are adapted (altered) to suit each new set of requirements. Component code must thus be available, in addition to knowledge to facilitate the altering of the code. The freedom to alter a component's code means that this approach can be very flexible, and can be constrained by legalities and contracts that are often packaged with third party components. Additionally, it can often be problematic to acquire a sufficient familiarity of how third party components work in order to alter them as needed. With white-box reuse, the reuse rate is likely to be high, as the potential flexibility available to alter the component to fit into the system is great, as is the resulting scope for reuse. The trade-offs are the amount of time needed to do this and the need to be familiar with the component code prior to its modification. White-box reuse is thus more common in in-house development, where access to component code can be assured without the complications of third parties. Therefore, it is necessary in white-box development to evaluate whether to reuse and adapt or to develop a new component from scratch.

Black-box reuse is where reusable components are not altered. The integrator does not need to understand, access or modify component code before components are integrated. Instead, such components conform to some specification, which can be utilised in the configuration and customisation of their reuse (Frakes et al, 2001; Parrish et al, 2001). However, the process of configuring the system to accept black-box components can be problematic. The flexibility with which such components can be used is also likely to be considerably more limited than for white-box reuse due to the predetermined way in which components are defined. Additional possible problems with black-box reuse include the need to make assumptions about and trust component quality, and the

range and number of components available for use. The availability of components that perform exactly as required is also an issue (Szyperski, 1998).

It is possible to split black-box use into two distinct categories: in-house and market. In the former category, components are selected that were developed solely within the same organization. This is likely not to be very highly used, due to the limited scope of components available to perform specific tasks relative to those on offer from other sources. The latter, however, is a considerably more popular reuse method. A wider range and quantity of components is available if the selection locus is extended outside the organisation, rather than being constrained to those available from within. This gives a greater chance of finding satisfactory components. With black-box reuse, there is thus also no time cost involved in altering the components themselves, although there is a likely cost involved in configuring components before integration. However, cross-organisation codes of conduct and legalities concerning the use of third party components through the marketplace brings with it additional complications.

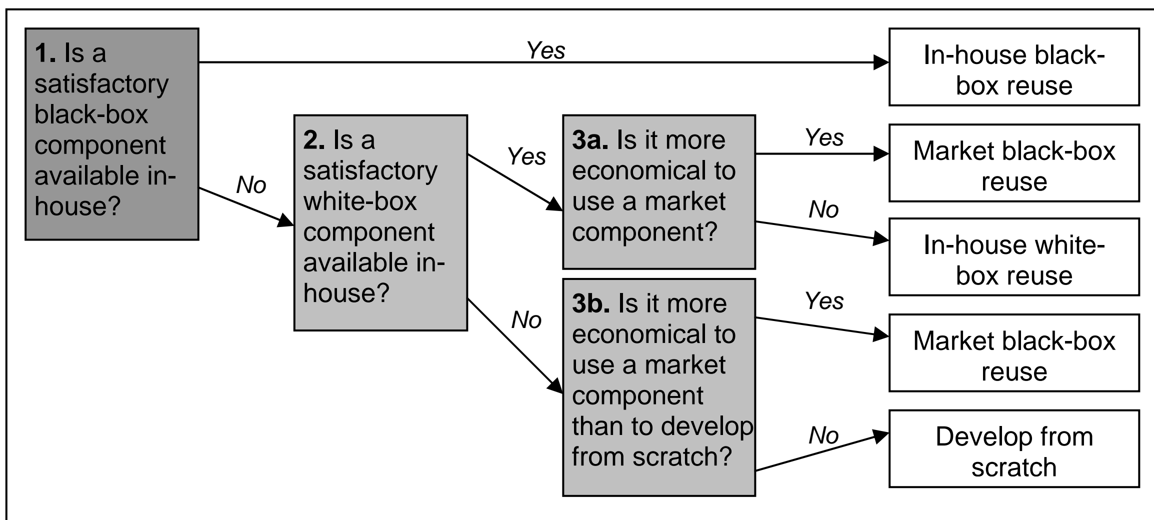


Figure 2 *Decision Tree for Component Reuse. Source: Ravichandran et al (2003)*

Bouchachia et al (2001) suggest a decision tree of which of the three reuse methods to use. A modified version is shown in Figure 2.

2.1.2 Adaptive Reuse

It can be useful to modify components for a range of different uses, if direct component substitution is not a viable option (Zaremski et al, 1997, Zarras et al, 1998). Basset (1997) describes the concept of adaptive reuse as:

"The process of adapting generalized components to suit various contexts of use."

There are different possible outcomes from locating such components in this way. Components may be identified using a query, which is essentially an expression of the developer's need (Mili et al, 1995). Hemer et al (2001) explain the different types of meanings of a 'query'. In its simplest form, it is a one-off comparison of the components against the requirements, the result of which is a list of components that satisfy these requirements. A query can also be modified and repeated, either to reduce the number of components returned by increasing the requirements or to increase the number by relaxing them. Successively refined and executed queries can also be used in two ways. Firstly, they each modified query can be run against the components returned from the previous one rather than from the whole range of components available, for example in a repository. The result would be to constantly reduce or 'filter out' the number of components to be decided upon. Secondly, each query can be run against all available components and the list of candidates

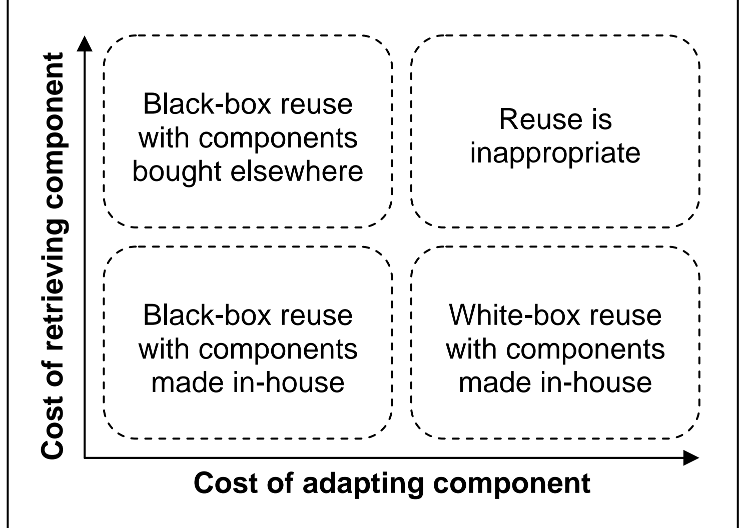


Figure 3 *White-box & Black-box Reuse against Costs*

added to those returned from the previous query. By modifying (but not necessarily refining further) each successive query, the number of components from which to choose increases.

Drawing from the points discussed so far, the usefulness of white-box and black-box reuse and adapting is shown in Figure 3, which is modified from Ravichandran et al (2003).

Hemer et al (2001) also describe the three possible matching mechanisms, which relate to the types of query explained above:

1. **All-match**, where a query[4] returns all of the one or more possible matches[5] that satisfy all the requirements of the query. The more times a query is refined and executed, so the smaller the set of components is likely to be. This is thus the most precise matching mechanism

2. **Some-match**, which is a more relaxed version of all-match. The same components are returned as would be in an all-match, with the addition of modules that satisfy some – or most, depending on the requirement restrictions – of the requirements. Unlike all-match, the number of candidate

[4] It is perhaps useful to note that by a 'query' we mean the end result of all successive refinements and re-executions of the query, since a query can be modified and re-run if the result (i.e. the quality of the match of components returned, if any) is unsatisfactory. In no instance, for the purposes of this work, will no components be returned, because it will be assumed that the query can be relaxed until at least one match is found.

[5] I.e. either exact and/or near matches.

components is likely to increase, as there is a greater and/or different range of criteria to compare against each time. This is the least precise matching mechanism

3. **One-match**, where a set of alternate queries are run in sequence against those returned from the previous query until only one component satisfies all queries run so far. This can be useful when the process of deciding upon a component to integrate is so automated that there is no place for a human party to the system to have the final decision on which component to choose from the candidates returned

All-match is useful when a number of functions are needed, rather than just one. This assumes that components can be combined (grouped) to perform a task.

2.1.3 The Problems of Reuse

Although reuse had its advantages, it also has its problems. According to Fischer et al (1997), the problems inherent in reuse strategies are as follows:

- Different reuse methods bring with them differing models of developing and organising the reuse of components. The result is the need to be familiar with each model being used
- Different components may be described in different ways and with differing detail. Some reuse models may thus only work on some components, given the description on offer
- More complex reuse models more precisely represent components, but are usually computationally expensive. Additionally, they tend to be more demanding on specifying requirements, resulting in complicated and time-costly usage overheads. Response times may also decrease due to increased effort (also Mittermeir et al, 2002)

- Reuse is only as good as the number and range of components that are available to reuse. A small library negates the usefulness of a software reuse strategy

- High recall, in which all or most matching components are returned, may suffer from over-the-top precision (returning correct matching components) of formal specifications (also Schumann, 1997)

- If human-readable documentation is supplied for specific components in adaptive reuse, the usefulness of the documentation will decrease. This is due to components being adapted for different purposes without constant revision (Mittermeir et al, 2002)

2.2 Distributed Systems

The DCBSE process, a result of the evolution of distributed systems and object-oriented concepts[6], provides the foundation for the study of distributed component-based systems. Such systems are often interchangeably referred to as distributed component-based systems, component architectures and component integration frameworks[7]. The term 'distributed systems' will be used in this document for uniformity and brevity to describe distributed component-based systems.

---

[6] For a good introductory background on object orientation, see Mili et al (1995) Section 4.3.

[7] Gomaa et al (1999) make the additional point that software systems can be composed into *application domains*, which they refer to as "a family of systems that have some features in common and others that differentiate them". This definition is analogous to that of the relationship of components to each other.

Distributed systems are often large and complex in structure, and the ease with which humans can cope with manual changes to such systems is decreasing (Baker et al, 2002; Kephart, 2003). There is also an increasing demand for systems that interoperate on multiple platforms, hardware technologies and software technologies, and in multiple programming languages, data formats and geographical locations, often simultaneously (Gokhale, 2002; McArthur et al, 2002; Stal, 2002).

Three elements need to be addressed in distributed systems in order to support the use of distributed components (Sousa et al, 2001). These elements are needed in order to achieve the goal of automation and thus reduce the burden on the human elements in the process.

1. The system's overall structure, including types and components used

2. Interface standards that describe these components' capabilities

3. A reusable infrastructure to aid integration

By describing the system structure and reusable infrastructure, the authors claim that it is possible to find components that can be adapted for use even if they do not conform to the requirements in their current state. This is termed *component mismatch*, or *fuzzy matching*.

Brown (1998) illustrates the following key requirements of a component-based system, which also apply to distributed systems, in order for it to be a success in terms of consistency, reliability and usability:

- Component behaviour should be defined both independently of the system and with regards to how it is to be implemented into the system

- The functionality of a component should be accessible through well-defined interfaces

- Component behaviour descriptions should be formalised for consistency of design and understandability

2.3 Repositories

The use of software repositories, libraries of components for (re)use in distributed systems, is widespread in Web technologies. According to Atkinson (1997), repositories are:

> *"collections of code fragments, operations, modules, systems, architectures, designs and specifications […] from which [repository] users can reuse the efforts of others during subsequent software development."*

Repositories need to be organised in such a way that they can be used in CBSE. The processes that are involved (Atkinson, 1997; Heineman et al, 2001) include:

- **Insertion** – how components are to be submitted, tested and put into the repository
- **Retrieval** – how component user is to access components from the repository
- **Adaptation** – how the component user is to know that a needed component is in the repository that can be reused effectively
- **Evolution** – how the component maintainer is to manage new and obsolete components and organise the components in the repository

An integrator can use repositories in two ways: by browsing and through retrieval[8] (Ebel et al, 2001). Browsing repositories manually can be slow and difficult compared to retrieval, particularly where distributed systems are large and dynamic (Bernstein et al, 2002). Furthermore, constructing libraries in ways that make browsing less complicated can themselves be time-consuming and expensive (Fischer et al, 1997). The remainder of this work will, therefore, concentrate on component retrieval mechanisms, which reduce these problems (Ebel et al, 2001). Retrieval depends on the availability of components and a component classification scheme, which in turn involves component descriptions. These areas will be discussed in the remaining sections of this chapter.

2.4 Components

There is no single standard definition of a software component. However, the literature reaches a general consensus about what a component is. (Note, therefore, that the definition of component here relates only to distributed systems within the context of the papers mentioned. Many completely different uses for the word 'component' exist outside distributed systems[9].) A widely cited definition, which satisfies most of the aspects of a component assumed by the literature, is Allen's (2001):

> *"a unit of composition with interfaces and context dependencies."*

---

[8] For an explanation of the difference between browsing and retrieval, refer back to chapter 2.1.

[9] For example, array elements in some programming languages are referred to as components. *Bell, D. and Parr, M. "Java for Students" 2nd ed., Prentice Hall, 1999, ISBN 0-13-010922-3, p231*

Allen describes an interface as a set of software services that provides access to component users, and a service as a set of functionality that supports some other function or provides information. This definition of a service follows a generic description of components that conforms to the majority of literature.

Interfaces are generally those parts of components, or separate objects relative to components, that provide access to the information necessary to use them (McArthur et al, 2001). Interfaces should be interoperable, so that they can be used with multiple software architectures, platforms, programming languages, hardware systems and other components. They may support legacy reuse and components re-engineering – that is, they may allow themselves to be modified when necessary (ibid., 2001).

Interfaces are widely regarded as being important and usually central in describing what a component does, and how it does it. Crnkovic (2002) describes an interface as “a specification of [a component's] access point,” and as “a collection of operations that specify a service provided by a component.” Interfaces are different from contracts in that, whilst contracts contain semantic details (*how* a component does something), interfaces only contain syntactic details (*what* a component does) (Maletic et al, 2001). It may also be desirable to avoid interfaces containing details about the component's implementation, for instance, exposing code or giving fine details about the operations of functions.

There are, however, definitions for components that are more specific and constrained than the all-encompassing ones offered above. McArthur et al (2002) describe a component as “an isolated element of a distributed composite system.” Meyer (1999) further affirms that a component “must be usable by developers who are not personally known to the component's author.” The definition provided by Meyer does not stand as a necessity for in-house developments, as the two actors, component developers and authors, can to be in constant contact to ensure consistent and reliable selection and

integration. However, the definition is important, as it is true of components that are liable to be freely distributed between globally distributed organisations. This is because the component developer is not necessarily the same person or party as the component integrator, nor is the developer necessarily available personally to provide knowledge concerning the use of a component (Allen, 2003; Szyperski, 1998).

2.4.1 Furthering the Component Definition

The details and purpose of various aspects of components need to be described. These aspects include (Atkinson, 1997):

- **Documentation**, which allows the actor or actors using components to understand what they do and how they are used
- **Scale**, which identifies the component's type (code fragment, module, architecture etc.)
- **Abstraction mechanism**, describing loosely how components are reused (i.e. with which software system architectures they are compatible)
- **Level of abstraction**, describing at what stage(s) of the component-based system's life-cycle the component becomes useful

These aspects deserve attention when components are described. However, components are only useful if they can be found. The retrieval process consists of three necessities (Bouchachia et al, 2001):

1. A way of describing components' functionality, structure and content

2. A way of representing a query that consists of the requirements of a component

3. A way of using the query to compare the requirements to the component's description

Components should also have a unique identity, a means of storing and/or advertising its state at any particular stage of use, and a defined behaviour (Arsanjani, 2002; Kontogiannis et al, 2003; Szyperski, 1998). Additionally, in order for components to be used successfully in a practical (as opposed to purely philosophical) environment, adherence to the following standards is of importance (McArthur et al, 2002):

- The infrastructure of the component should be consistent, standardised and describe functions and properties

- There should be explicit, well-defined but not too numerous context dependencies

- The interface should support components requiring other components and allow the appropriate communication to allow this

- Interoperability should be addressed, in terms of platforms and languages

- Integration should be dynamic, seamless and support versioning

The system supporting component integration should also facilitate understandability and be sufficiently abstract to allow for extension and future enhancements (ibid., 2002). Further characteristics important in the composition of components include the following:

- Components and component versions should be clearly distinguishable from each other and their environment (Crnkovic, 2002; Meijer, 2002)

- Components should communicate via interfaces for functional properties, and contracts for non-functional properties such as context dependencies (Crnkovic, 2002; Heineman et al, 2001; Kontogiannis et al, 2003)

- Interfaces and contracts should be physically independent of, but closely related to and well-defined by, their respective components (Crnkovic, 2002)

- Descriptions should be as easy as possible to understand, to reduce and minimise as far as possible the severity of the learning curve (Rodden et al, 2002)

Kephart et al (2003) add to the characteristics of components by suggesting further details of what component composition should address in the future:

- **Self-optimisation** - the component should be able to perform optimally, whether by manual tuning of parameters (if such information about the detail of the component is available) or automatically

- **Self-healing** - the system maintainers should be able to identify, trace and determine the reasons behind failures of components and automatically correct these problems if possible

- **Self-protection** - the component should provide a measure of security to protect the system, or its constituent components, against attacks

- **Upgrading** – components, and thus systems, should be able to upgrade themselves when new versions of the software are available

The ISO/IEX standard 9126 sets out a series of standard attributes that reusable components should have for the purposes of measuring quality (Torchiano et al, 2002). They are presented and defined as used for the purposes of this document.

Term	Definition
Functionality	The capability of a component to provide functionality meeting needs when used under certain circumstances
Reliability	The ability of a component to provide functionality reliably with a minimum of faults
Usability	The ability of a component to be understood and used with ease
Efficiency	The performance measurement of the ability of the component to perform some given functionality
Installation	The process of integrating components into a framework
Integration	The process of combining software components, hardware components or both into a framework
Maintainability	The ease with which the component can be modified, analysed and tested when necessary
Portability	The ability to conform to multiple platforms and systems or to be adapted and installed to be supported by other platforms and systems

In addition, the following definitions of other aspects of components and component will be used:

Term	Definition
Matches	(Of components) “satisfies”; “meets”; ‘is equivalent to’; “interacts properly with” (Zaremski et al, 1997)
Query	To use the explicit description of the requirements to search for a suitable component or components (Rodden et al, 2002)
Retrieval	Given a set of requirements, to find a set of components whose combined behaviour satisfies those requirements (Atkinson, 1997)

In order that the system can operate once a component is described, a number of steps are necessary in order for a component to be chosen and used (see Figure 4) (Heineman et al, 2001; Kephart et al, 2003):

Action	Component supplier	Component user
Location	Advertise details about the geographic location of the component	Gather details about the geographic location of the component
Negotiation	Agree with how the component is to be used. This may include such issues as price, priority of service, nature of payment, type of license and other strategies and protocols	
Provision	Record details about the negotiated agreement for component use	
Operation	Provide the component or component functionality	Use the component or component functionality
Termination	End the agreement by terminating the negotiated contract of component use. This might never occur, or may occur due to the supplier going out of business or the component's use no longer being needed for example.	

Figure 4 *Stages of Communication between Component Supplier and User*

2.4.2 Overall Component Definition

For the purposes of this work, the following assumptions about components are made:

- **Components are self-containing entities** which can preferably be distributed freely in terms of system, geography, platform and programming language

- **They have at least one functional ability and/or piece of knowledge**. Its purpose should be made clear to the third party or parties locating, evaluating and using the component

- **They define functions and properties** so that a fair assessment of the component's functionality can be made by third parties

- **They have standard, associated interfaces, contracts and context dependencies** so that the component's description can be defined in a recognisable and interoperable way

- **They must be readily accessible to third parties** that require components or component functionality[10]

2.5 Component Descriptions

Components need to be described in order to help integrators find suitable components to (re)use and suppliers to describe their products and distinguish them from others. These descriptions must be sufficiently flexible to describe appropriate information precisely and to support the architecture in which it is used. The description languages that exist have emerged in an attempt to solve some of the issues that have arisen from poorly described components. The process of and rationale behind the evolution of description languages, and an account of the advantages and disadvantages of some of the most notable ones, is given here.

Previous (textual) description methods, which involved commenting and describing source code, are becoming increasingly outdated as the number of components available and the complexity of component-based systems increases. The need for computational speed has similarly increased, whilst the amount of time system integrators wish to spend understanding code decreases as the amount of source code gets larger (Mittermeir et al, 1998).

The evolution of description languages is focused less on textual description methods and instead towards *natural language* and *formal specification* descriptions. Component suppliers need to establish what components can do, how they do it and how they are to be used – i.e. they should be *classified* – so that reusable components can be appropriately identified and used (Liao et al, 1999). Integrators need to establish the requirements for an appropriate component. They use these requirements in natural language and formal specification descriptions as a basis for *querying* a given component list for suitable components[11]. Both component descriptions and requirement descriptions thus need to be compatible in order to find a suitable component match to integrate into the system. Thus, there is a need to address the issue of making the integrator's intent, and the description format used to describe that intent, compatible and clear (Mittermeir et al, 1998). Additionally, the retrieval method, using the description, should be able to operate in a manner of efficiency sufficient to be both flexible and timely (Damiani et al, 1996).

Zaremski et al (1997) refer to this as *specification matching*[12]. Their definition, which also explains the use of the word *substitution* within the context used in this document, is:

---

[10] The entity may, however, be subject to constraints such as legal agreements and licensing.

[11] A useful definition of a query is "an explicit description of the user's current requirement" (Rodden, 2002).

[12] Specification Matching is covered in more detail in section 2.5.2.

"a way to compare two software components. In the context of software reuse […] it can help determine whether one component can be substituted for another or how one can be modified to fit the requirements of the other."

Before an overview of some of the main approaches to component descriptions is given, definitions of some of the technical terms used in this document are merited for clarity. A brief description of the evolution, use and issues of previous and current description methodologies will then follow.

Term	Definition
Classification	The method of capturing information about components that can be used to query and match components with the requirements
Interoperability	The ability of a description language to operate irrespective of platform or programming language
Property	A single descriptive object to which a value is assigned
Requirements	Specifications and preferences that a system composer must decide upon before searching for a component. Such details may include functionality, quality, supplier and/or precision

2.5.1 Keyword-Based Descriptions

Keyword-based descriptions involve component developers associating descriptive keywords with a component[13]. The requirements of the system into which components are to be integrated are then

[13] Various such description techniques existed before component-based systems, which describe documents passed over networks (Henninger, 1997). One example is Latent Semantic Indexing (Maletic, 2001), which records the meanings of words and passages. Such a technique is, however, inadequate for describing components.

summarised by the integrator to produce a set of keywords. If these keywords match with those of a component, then that component is deemed to be a potential component to use. This description mechanism is the result of an evolution of the idea that objects can be identified by underlining nouns. This can be too informal and imprecise for component use in the real world (Chen et al, 1993; Damiani, 1997; Levi, 2002). The keyword-based method can also be improved by prioritising keywords, in the event that multiple candidate components are found, or by increasing the number of keywords to search, which typically improves the precision (Bernstein et al, 2002; DeRoure et al, 2003; Liao et al, 1999).

However, the precision of this method can present a problem, as keywords capture the syntax of the requirements, but not the semantics. As exemplified by Khayati et al (2002), it would not be possible to specify the query “select all the components using Java technology which implement the X interface.” Additionally, a badly chosen set of keywords, however well intentioned the keyword compiler is, will produce a bad result, and it can be difficult or even impossible to get a meaningful match. A real-world example of misusing keywords would be attempting to sell ‘Ginness merchandise’ on E-bay: Given the misspelling of Guinness, the merchandise would be much less likely to reach the whole intended audience, even though it is a much a feasible search candidate as one whose description is spelled correctly. This issue can be addressed by introducing a standard, finite predetermined set of keywords (referred to by Damiani et al (1997) as a *descriptor base*) to choose from. However, this can still result in misinterpretation of the meaning of the keywords upon choosing an appropriate description.

Keyword-based descriptions have proven insufficient as distributed systems become more complex and the range and quantity of components increases. A natural evolution of the keyword-based approach has proven to be *table-based* description methods.

2.5.1.1 Table-Based Descriptions

Table-based descriptions, an enhancement on the keyword-based idea, describe pairs of attributes and values. The attributes are standardised to make comparison easier. The component selection process is similar to the keyword-based method: Requirements are described using the standard attributes, and the search attempts to match attribute-value pairs. In this method, a finite set of descriptions are created by the component developer, making it easier to structure descriptive information and thus minimise the risk of misinterpreted descriptions. This is because 'linguistic noise' is filtered out (Damiani et al, 1997). It reduces the risk of *infosmog* - the condition where too much information is gathered to be of use in a practical amount of time by the integrator - which can affect description methods where the number of keywords is not limited (DeRoure et al, 2003). However, it still does not cater for the broader aspect of the component or of the necessary system, namely the semantics (Bouchachia et al, 2001; Maletic et al, 2001). Additionally, the keywords used to describe each attribute either may be non-constrained, leading to a more natural but less predictable way of describing, or constrained, limiting the expressive ability of the method (Penix et al, 1999).

There are many variations on this theme. One such variation, explained in Liao et al (1999), is the “verb-noun + weight” scheme. In this scheme, function keywords are described in such a way that two keywords, one verb and one noun, are coupled with a weighting ranging from “very high” to “very low” in its relevance to another verb-noun pair. An example would be “resize window,” where the relevant operating system library would score a rating of 5 whilst a component described as “read

value" would be given a rating of 0. Multiple keywords can be grouped together, and in this way weightings of a series of verb-noun pairs together, rather than just one pair, can be measured and evaluated (Rodden et al, 2002; Torchiano et al, 2002). Many other methods using weighting measures of various complexities exist to improve the precision with which components are queried with the requirements, although few are used in reality due to a range of disadvantages unique to each one[14].

Table-based description methods are an improvement over the simpler and less precisely descriptive keyword-based approach. However significant improvements can be made, particularly in repositories in which there are large quantities of attributes and values, by organising groups of related descriptive information into hierarchies. This approach is referred to as *enumerated classification*.

2.5.1.2 Enumerated Classification

In this classification method, component properties are organised into categories, which are in turn organised into hierarchies. In other words, a component is placed into a category, which may contain one or more nested sub-categories that further describe aspects of the component. This way, a predetermined ordering and organisation of categories can be queried more easily by an actor that understands the rules about how the information is organised, rather than a 'flat' collection of descriptions where a large number of attributes are defined. The usefulness of this method would therefore increase with the number of attributes (Henninger, 1997).

[14] For example, Fox (2003), Hemer (2001), Henninger (1997)

2.5.2 Structural Matching Descriptions

The previous techniques attempt to match components given some form of description relating to the content and abstract functionality of a component, which can be inadequate due to the focus of being exclusively on the syntax with which functionality is described (Chen et al, 1993). "Structural matching" is an attempt to represent the structure of a component's functionality. (Note, however, that the term "matching" used here refers only to the name of the description method. The method itself does not match components within the definition given in section 2.4.1.) Two notable examples of this approach are *signature matching* (which is described in section 2.5.4.1, following an explanation of fuzzy matching techniques) and *specification matching* (see below) (Khayati et al, 2002).

"Specifications," or logical predicates, have also been captured and used as an attempt to describe components formally, i.e. using standardised descriptions. The use of predicate equivalence can thus match components by matching the pre- and post-conditions[15] of each function that a component offers to those specified as requirements. This is known as *specification matching* (Chen et al, 1993; Fischer, 2000; Penix et al, 1997; Penix et al, 1999; Zaremski et al, 1997). This means that specification matching is domain-specific, relating to particular use paradigms and behaviours rather than descriptions of attributes relating to use. It is for this reason that specification matching is an example of what is known as a *behavioural description* (Khayati et al, 2002). Various specification languages

[15] Pre- and post-conditions can more specifically be defined as "logical functions of the respective parameters" (Fischer et al, 1997).

such as Z can be used to describe a component, and programs called *theorem provers*[16] identify mathematically which components are (best) matches[17] (Khayati et al, 2002).

A wide and potentially non-exhaustive quantity of component semantics can be described through structural matching approaches. Attributes that can be formally specified and measured (Torchiano et al, 2002) include:

- *Market share* (popularity of this type of product)
- *Performance* (ability to match scalability requirements)
- *Safety and security*
- *Reliability* (fault tolerance)
- *Hardware and software requirements*
- *Change frequency* (number and frequency of change of component versions)
- *License type* (for example, commercial or freeware)

[16] An extension of using theorem provers is *rule matching* (Hemer, 2001), where queries are structured as "proof goals" and the patterns made by the theorem prover are made up of definitions and theorems.

[17] More specifically, a "match is where the precondition of the library component is weaker than that of the query, and the post-condition of the library component is stronger" (Hemer, 2001)

- *Cost of use* (for example, price per use or price per license)
- *Conformance* (standards met)
- *Domain specificity* (if domain-specific, list domains to which the component is applicable)

Formal specifications need to be concise and precise to be of any benefit, and machine-parsable to make the search for components in repositories easier on the (human) actors responsible for the search, if any (Hemer et al, 2001). However, a problem with formally specified and automated components is the difficulty in describing components so that an integrator searching for components will recognise and understand the same description context and conduct the search using the same property names. In other words, it is unlikely that the person searching for a component, or some given functionality, will compose a query that matches exactly the descriptions of the components on offer (Damiani et al, 1997). One popular attempt to get around this problem has been to form descriptions that accept a wider range of descriptions of the same functionality. This way, a query using one way of describing what is needed can be related to another way that is similar or identical (Bouchachia et al, 2001). This school of thought is widely referred to as *fuzzy matching*.

2.5.3 Fuzzy Matching and Classification

It is not likely that a component exist that conforms precisely to the expectations specified in the requirements (Atkinson et al, 1994; Zaremski et al, 1997). Fuzzy matching allows the facility to match the requirements of the system to a component that does not necessarily match exactly all of the criteria. Thus, a degree of estimation is given in 'matching' components to requirements. There are many different approaches to fuzzy matching, and some of the most widely cited ones are *faceted classification* and *deductive retrieval*.

2.5.3.1 Faceted Classification

The “faceted” keyword-based approach is intended for use with small components that perform one simple task (Baruchelli et al, 1997; Prieto-Díaz et al, 1991; Liao et al, 1999). The faceted classification scheme comprises a description of six single facets (properties), so the procedure for matching components is simpler than methods allowing many entries per attribute (see Figure 5). Each facet has a name, which is chosen from a range of relevant terms stored in a word database called a *vocabulary*, and associated value (Bouchachia et al, 2001; Khayati et al, 2002).

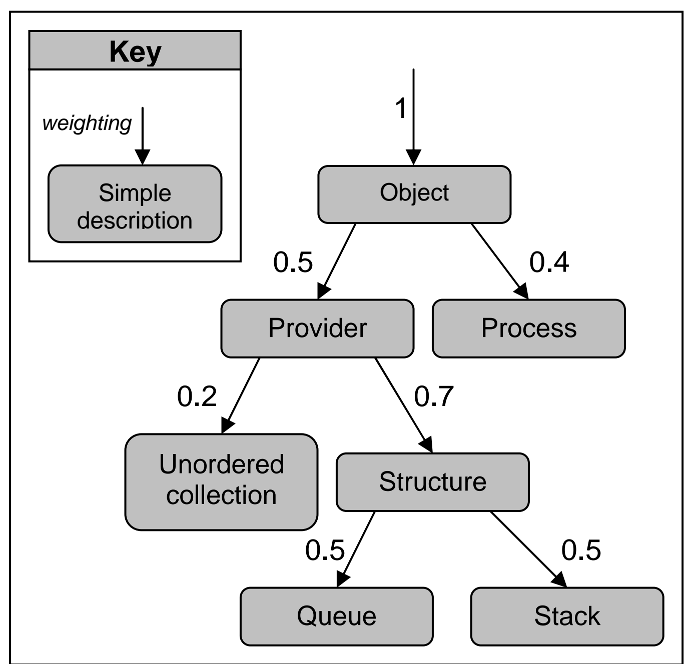


Figure 5 *Example Faceted Keywords & Weights*

Baruchelli et al (1997) exemplify this schema and represent it in a data-flow diagram similar to that shown in Figure 5. The ‘weighting’ values represent the order of preference for one property over another, in cases where a property that is reached earlier in the hierarchy in turn points to more than one possible property. The weightings therefore represent the likelihood that a component property will be equivocal to that specified in the requirement set of properties, 0.0 representing ‘definitely not’ and 1.0 ‘definitely’. However, the weighting values themselves are arbitrarily chosen to reflect the (im)precision of the match, and can consist of a value between 0 and 1, 0 and 10 or even “very high,” “high,” “low,” and “very low,” for example. The former weighting method is used in this example.

Three “functionality” attributes discussed in the Liao general component description are as follows.

1. **Function**, i.e. the usable functionality that the component provides

2. **Object**, i.e. components and resources that this component uses

3. **Medium**, i.e. in which situation the function is carried out and the results stored

Therefore, the component of a desktop calculator that multiplies two numbers together might have the function “multiply,” the object “maths library” and the medium “text-field”

In addition, reusable components contain three additional “environment” attributes (Liao et al, 1999):

1. **System Type**, i.e. the environments in which the use of the component can be employed

2. **Functional Area**, i.e. why the component should be used

3. **Setting**, i.e. the location of the system that should use the component

In the previous calculator component example, System Type might be “C++”, Functional Area “multiplication” and Setting “mathematics”.

The scope for matching is limited, as the actor describing the component can choose from many keywords meaning the same thing, some of which might be missed in a search. For example, a set of search criteria identical to those for the calculator component whose Functional Area was described as “duplication” rather than “multiplication” might miss this component. Faceted matching attempts to overcome this by examining the “closeness” between two descriptions. Thus, related keywords, such as “duplication” and “multiplication,” could be coupled together and used synonymously in the search. Additionally, the extension of the faceted scheme to allow multiple entries for each of the six attributes, as introduced by Liao et al (1999), alleviate the problem of limited keyword descriptions.

The faceted classification technique is used to store facets that are mutually exclusive, so that it is possible to distinguish the facets of one component from those of another in order to obtain a 'best' match. The maintenance of component descriptions is also made considerably easier because one facet can be modified without having to redefine another (Henninger, 1997).

2.5.3.2 Deductive Retrieval

Fischer et al (1997) introduce the idea of "formal specifications," and investigate its use as a means of improving scalability, efficiency, reuse-friendliness, library organisation and integration. In their view of the structural matching technique, each actor states a "contract." Contracts consist of component requirements, specified by the client, and a declaration of how a component is to be used, specified by the component provider, in terms of pre- and post-conditions. Negotiation then takes place based on the contracts of each side and the compatibility between the conditions, using a proof provider to ensure that compatibility (see Figure 6). The process of defining formal contracts tailored to particular actor roles and their relationships is called *deductive retrieval*.

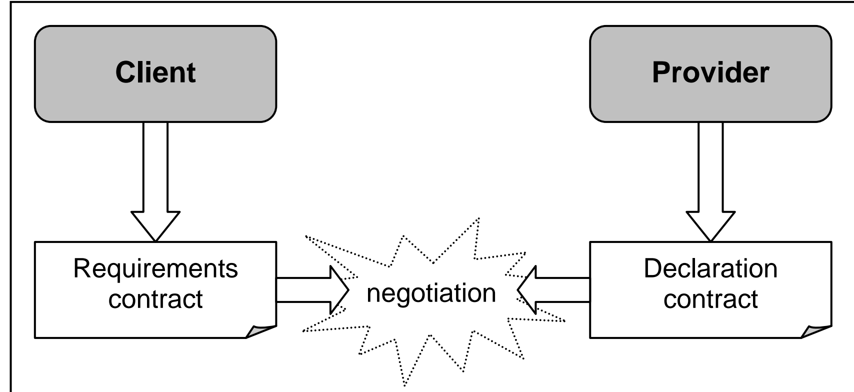


Figure 6 *Contract Relationship between Client and Provider*

Deductive retrieval also addresses the issue of *granularity* - that is, the size and nature of a component and its make-up (Bennett et al, 2001; Damiani et al, 1997; Thomason et al, 2002). There may be no or many components that can be considered appropriate matches, and the contracts may be relaxed or tightened to control the number of components being reported as matches. The stronger the contract is,

the lesser the benefit of a "proven" match but the more likely it is that a component will be found that satisfies those requirements. However, more relaxed matches increase the likelihood that manual compatibility checks will be necessary.

The need to understand formal descriptions is often a lengthy process. The limitation of descriptions in the vocabulary can also lead to a limitation of how component usage (in terms of how they are used and what they do) can be expressed (Bouchachia et al, 2001). This has given rise to the idea of describing them in other ways, namely in the form of *non*-functional descriptions.

2.5.4 Non-Functional Descriptions

It is important to capture semantic (non-functional) descriptions of the *context* in which functionality is supplied, in addition to the syntactic (functional) descriptions of *what* is supplied. As explained in DeRoure et al (2003), the expressiveness needed for the reasoning process that precisely refines component matching can be captured in *ontologies*. These ontologies conceptualise functional properties and show the relationships between them - i.e. the way in which components behave. DeRoure et al (2003) outline three ontologies that are of relevance here:

- **Domain ontologies**, which use a predefined set of keywords for describing component design features

- **Quality ontologies**, such as expected error rates, qualifications and certification, and an expected degree of accuracy

- **Value ontologies**, such as the cost and scarcity of content, functionality and data

In addition, there are a number of aspects of components that can be described, including (Vitharana et al, 2003):

- **Synonyms** (as has been discussed): Using related descriptive words to describe functional aspects of components
- **Roles**: How a component is intended to be used
- **Business rules**: Additional constraints to describe processes and functions to match the component to the business's requirements
- **Function/task** (as has been discussed): Describing, in non-functional terms, the purpose of a component's individual methods
- **Element/part**: Other fundamental aspects of a component, such as variables
- **Action/event**: A description of what actions are expected within the component and what corresponding results (events) can be expected
- **User**: The intended users of the component

The information contained within each ontology should be presented in a clear, unambiguous way, which can present a problem due to the amount of ways in which the information can be compiled and expressed. As with the problem with table-based and keyword-based descriptions, liberalisation means a large volume of data to sift and possible incompatibilities with and misunderstandings of the information given. Furthermore, constraining means a restriction of the (here semantic) accuracy with which information can be given. Non-functional descriptions have different meanings depending on the circumstances in which they are used, which threaten the consistency of those descriptions (Damiani et al, 1997). However, it is the removal of ambiguities, both with the information presented and with the component's purpose and use, which non-functional descriptions aim to solve.

The above methods assume that it is appropriate only to match requirement keywords with keywords used in component descriptions that are an exact, word-for-word match. Considerable work has been undertaken in the field of *fuzzy matching*, where components are matched to descriptions whereby words match *semantically* rather than necessarily syntactically. One of the simplest ways in which this is done is to store synonyms of related words in a database, or *thesaurus*. This means that groups of associated words are deemed to be matches, even if the words themselves are syntactically different. Each synonym can be associated with its own weighting, which judges the amount of precision with which it is supposed to match a related word, called *fuzzy relationships*. Component query languages use fuzzy algorithms based upon these keyword associations and weightings. They can thus be used to attempt to match up component descriptions with requirement descriptions without the need for a component to match exactly in the sense of non-fuzzy matching methods[18] (Damiani et al, 1996; Damiani et al, 1997). However, weightings would necessarily be arbitrary, since there is no way of saying objectively how precise a semantic match one word is to another. Therefore, the usefulness of the method has to be disputed, as the quality of results is likely to vary depending on the weighting method used and thus the appropriateness of weightings. Additionally, it is unlikely that a "perfect" match would be found, because semantic consistency between those needed and those supplied by components cannot be ensured.

---

[18] As Mittermeir et al (2002) put it, "to a person with a nail, any tool looks like a hammer." In other words, it doesn't matter whether the component is an exact match as long as it can get the job done.

2.5.4.1 Signature Matching

The previous description methods focus on sets of attributes that describe functionality. Zaremski (1995; et al, 1997) introduce the signature matching method. In it, signature descriptions are an attempt to avoid or at least minimise the need for functional descriptions by describing instead a component's *type information*[19] (Zaremski et al, 1997).

The signature matching approach focuses on describing the data types of the component's interface. Typically, the type (e.g. Boolean, integer, character), range (e.g. 1 to 10, 'a' to 'z') and number of types in the interface are recorded. These data act as "keys" that can be applied when searching for a component that is compatible with the requirements (Fischer et al, 1995). This is called *function matching*. Additionally, "features" can be asserted to identify whether the types are simple – that is, native to a programming language or usage context, constructed, user-defined or functions, for example[20]. This process is called *module matching* (Fischer et al, 1995, Zaremski, 1995).

These two kinds of information are represented as the component's *signature*. The process can be completely automated, assuming that information regarding a component's interface types is available (Hemer et al, 2001; Khayati et al, 2002).

[19] A mathematical definition of signature matching is "the universal relation defined over the cross product of all types" (Mittermeir, 1998).

[20] An alternate definition of a signature is a set of sorts (types) and operations (features) (Chen, 1993).

The matching approach taken by Fischer et al (1995; 1997) is, at its simplest, twofold. It attempts to utilise the advantageous aspects of both “classical” (i.e. functional, syntax-based, or keyword-based) methods and tie in the gains that can be made through semantic-based methods. The types, as with signature matching, and pre- and post-conditions, as with specification matching, of each component are recorded to create a *VDM specification*. During the search process, signature matching is used either to match the types of each component to the component being replaced, or the types specified in the requirements. The list of candidate components that is created from this phase of the search process is then passed into a further component filter by matching the pre- and post-conditions. The work claims that the advantages include the possibilities to combine multiple search methods, both functional and non-functional, according to the levels of match precision required.

2.5.4.2 Static Behaviour Sampling

Mittermeir et al (2002) set out one the most advanced of those description methods described above. In this method, which they call *static behaviour sampling*, pairs of sample inputs and associated expected outputs are stored with the component description. In addition to using input and output data through the signature matching technique, the retrieval mechanism gains details of components by using this additional data to test a component’s fault tolerance. Therefore, the method does not rely on test data to describe component functionality, only to reveal any faults that a component may have. The test pairs are “striking examples” that test the limits of particular functions to ensure the degree of quality. They can also be used to distinguish between different components. A simple example of what the pairs of test inputs and associated outputs might look like and represented as is shown in Figure 7.

Samples	Substring	Prefix	Equal	Greater	Smaller	Longer	Shorter
<'a';'b'>	F	F	F	F	T	F	F
<'';'a'>	T	T	F	F	T	F	T
<'a';'ba'>	T	F	F	F	T	F	T
<'';''>	T	T	T	F	F	F	F
<'a';'a'>	T	T	T	F	F	F	F

Figure 7 *Example Pairs of Test Data for Static Behaviour Sampling, in Tabular Form*

2.5.5 Precision and Recall

Two intrinsic aspects of description languages are their ability to describe precisely and suggest candidate alternate components. Figure 8 shows the approximate relationship between the specification-based retrieval methods that have been discussed with these aspects in mind.

Keyword-based methods do not typically locate many candidate components due to their simplicity (recall), and the degree with which the candidate components suit the requirements is also relatively low. Table-based methods tend to have the same level of recall but provide increased precision over keyword-based methods due to its attribute pairing and weighting mechanism. Specification-based methods tend to have higher recall and precision due to the way it creates hierarchies of properties. Enumerated and fuzzy matching methods promise higher recall, although the degree of precision often depends on how well properties and weightings are chosen (Bernstein et al, 2002).

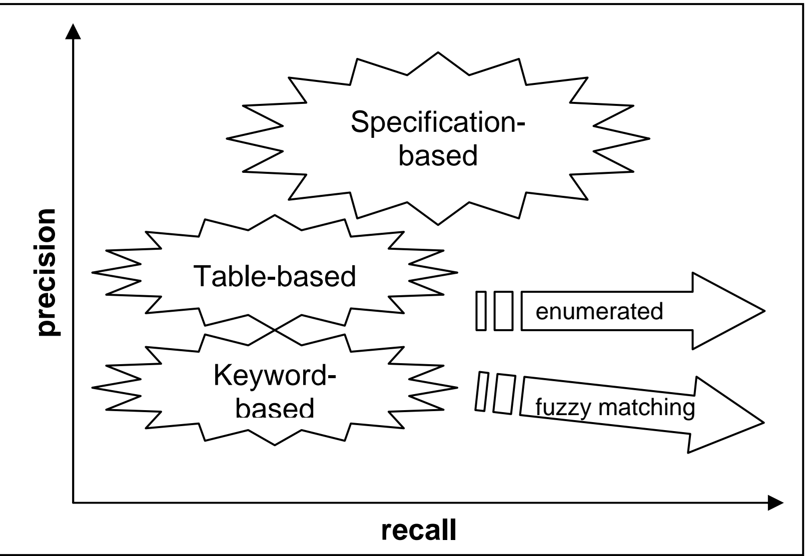


Figure 8 *Comparison of Precision and Recall of Retrieval Methods (modified from Bernstein (2002))*

2.5.6 Limitations

From the literature, a series of limitations has been identified relating to each description method. The ones that appear most frequently are summarised below (Liao et al, 1999).

2.5.6.1 Functional Description Limitations

- Keywords to use either too small or too large, leading to over-simplified or sometimes impossibly lengthy searches
- Do not fully or completely precisely describe component functionality
- Often suffer from problems when different versions of the same component exist which may both have the same signatures (Mittermeir et al, 2002)
- Table-based descriptions do not capture information in a range that increases precision, compared to semantic descriptions (Bernstein et al, 2002)

2.5.6.2 Non-Functional Description Limitations

- Ambiguity due to there being more than one meaning for each term in the definition, and there being a consequent degree of uncertainty and doubt about the meaning or intention of a statement (Bouchachia et al, 2001)
- Fuzziness of descriptions between components in inexact matching can lead to unclear distinctions between them (Bouchachia et al, 2001)
- Producing semantic information is commonly a time-consuming and expensive task (Maletic et al, 2001)

- Difficult to measure semantic attributes in a meaningful way (Torchiano et al, 2002)

2.5.6.3 Description Limitations Common to Both

- Matching granularity. Course-grained components need to be adapted prior to reuse, while fine-grained components make reuse less effective (Hemer et al, 2001; Mittermeir et al, 2002; Thomason et al, 2002)

- Versioning. Producing different versions of components may cause confusion during retrieval. Additionally, backwards compatibility inherently bloats the size of each component as it evolves to meet new needs (Meijer, 2002)

2.6 Some of the Emerging Fields of Distributing Systems

A number of developments have arisen from distributed component-based systems and the needs originally identified by Brown (see section 2.2). Some of the emerging fields are Grid Computing, Web services, Software as a Service (SaaS) and Commercial-Off-the-Shelf (COTS). These will be covered briefly to illustrate the approaches taken in light of the developments in distributed systems. A distributed architectures case study, CLARiFi, will then be illustrated.

2.6.1 Grid Computing

Grid Computing is so called because of its

> *"analogy to a power grid that provides consistent, pervasive, dependable, transparent access to electricity irrespective of its source."* (Baker et al, 2002)

Grid Computing covers a broad range of distributed data storage, sending and receiving issues, spanning a large scope of organisational types (ibid., 2002). As a result of this wide scope, Grid Computing stresses the importance of domain independence, support for a range of distributed technologies (heterogeneity), scalability and adaptability. This gives rise to the possibility of a range of service types that distribute functionality and information in a number of ways. Using these services, systems following the Grid Computing infrastructure can be constructed in a (theoretically) seamless way. The services that Grid Computing makes possible include (Baker et al, 2002; DeRoure et al, 2003)

- **Computational services**, which can securely schedule, execute and/or allocate functionality on large amounts of data in distributed libraries and other resources
- **Data services**, which give access to information and the ability to manage this information
- **Application services**, which provide the ability to manage and coordinate services
- **Information services**, which identify the ways in which information – for example, the results of using computational, data and application services – is to be presented meaningfully, stored, accessed, shared and maintained
- **Knowledge services**, which manage the way that information is retrieved, used, published and maintained

DeRoure et al (2003), further enhance this perspective, offering that:

> *"there should be a high degree of automation that supports flexible collaborations and computation on a global scale … this environment should be personalised … and should offer seamless interactions…"*

and that Grid Computing should offer the ability to ensure:

- **Ownership** of content and capabilities of that content
- **Conditions of use**, including terms and conditions and legal contracts
- **Transparency** of content discovery and access
- **Communities** of organisations and collaborators which can control content distribution and use and incorporate privacy from external organisations
- **Security**, including authentication, encryption and privacy
- **Reliability**, which relates to quality-of-service and trustworthiness

Grid Computing services can also be viewed as agents that interact, producing and sharing knowledge according to predefined communication methods, or protocols. Thus, the ability to interoperate effectively also plays an important part in ensuring a system that successfully implements the Grid Computing infrastructure.

2.6.2 Web Services

The components that are composed, advertised, supplied, located, retrieved and utilised can be viewed as services. Actors may supply and use services in a business context. A number of types of services can be offered (Bernstein et al, 2002):

- Applications, such as Web services
- Components
- Repositories

- Advice organisation services, which can offer such advice as working practices and recommended services, and legal information

Although the latter three are commonly seen in distributed systems, the concept of Web services has gained considerable attention.

The World Wide Web Consortium (W3C) describes a Web service as

> *"a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered…" (W3C, 2002)*

A wide aspect of the DCBSE paradigm is covered by the Web services infrastructure, including interface and implementation descriptions, service composition and integration, agreements and contracts, publication, and discovery (Kontogiannis et al, 2003). A strong focus is on the standardisation of these aspects. For a service to be integrated, its interface is first described using Web Services Description Language (WSDL). This description is then used to *bind* a service, i.e. describe how it interacts with other services. Finally, the Web service is published for discovery by integrators (Kreger, 2003).

Web services are built on the idea of layers, which relate to the three roles that services play (see also Figure 9) (Kreger, 2003):

- **Interactions**. A transport layer allows communication, and a packaging layer defines how the information is packaged prior to communication. Both are based on XML (and often SOAP and HTTP, which are more structured)
- **Descriptions**. A description layer is necessary to specify the information being communicated. Additionally, interfaces define the information necessary to describe how particular Web services

interact with each other. Such information can include ownership, security, cost and quality. Other layers describe related services, the order in which operations are carried out, a service-level agreement layer that captures performance, cost, metrics and thresholds, and a business-level agreement layer that captures legally contractual constraints

- **Discovery**. Technologies should be available to allow services to be published (made available), and thus discovered using the descriptions given with each service

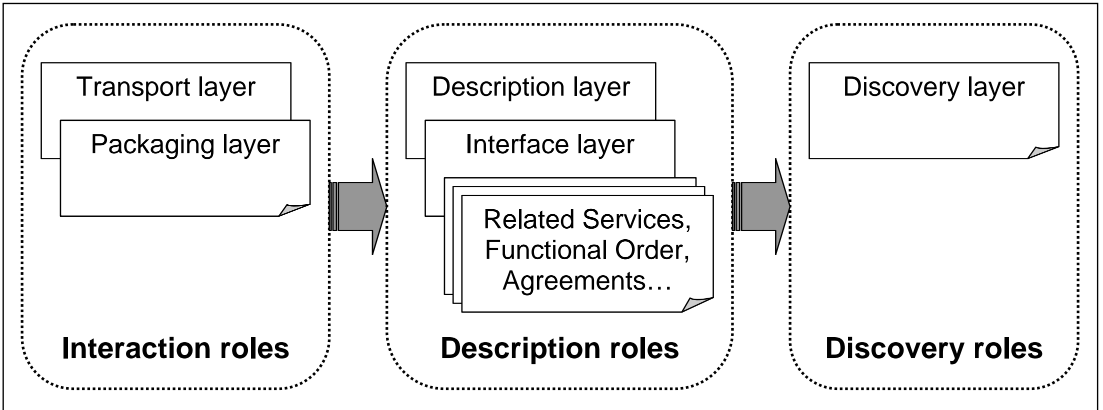


Figure 9 *The Roles that Services Play in the Web Services Infrastructure*

Current Web services are based on standards such as the Extendable Mark-up Language (XML)[21] and Simple Object Access Protocol (SOAP)[22]. They are thus interoperable. The scope of the interoperability of Web services can be extended by using standards such as the Web Services

[21] XML Web site: http://www.xml.org/.

[22] SOAP Web site: http://www.w3c.org/ . SOAP 1.1 can be found at http://www.w3.org/TR/SOAP/.

Description Language[23] (WSDL) and Universal Description, Discovery and Integration[24] (UDDI) to communicate between services (Fremantle et al, 2002; Stal, 2002). Note that it is more appropriate to use the word 'extended' to describe this process, rather than 'modified'. 'Modified' suggests that the existing infrastructure is changed, whilst in fact it is built upon.

2.6.3 Software as a Service

The concept of *Software as a Service* (SaaS) concentrates on the supply of components as services. Services can be assembled and combined at any time during the running of the distributed system, usually immediately before they are needed, something that is termed ultra-late binding (Turner et al, 2003). The proposed service model is different from those used currently, which typically contain the distributed system software and the transport medium, for example Sun's Java 2 Enterprise Edition (J2EE) and Microsoft's Distributed Component Object Model (DCOM). Such

Service Layer
Applications created 'on demand' from smaller services)

Service Technology Layer

Service Transport Layer

Figure 10 *A Proposed SaaS Service Model (taken from Turner et al (2003))*

[23] WSDL Web site: http://www.w3c.org/. WSDL 1.2 can be found at http://www.w3.org/TR/wsdl12/.

[24] UDDI Web site: http://www.uddi.org/.

models are made up of services that were available before the distributed system was composed, meaning that the system would have to utilise what is already available, i.e. they are supply-led.

A proposed SaaS model, shown in Figure 10, involves a demand-led architecture, in which services are composed as requirements arise for given functionality (Bennett et al, 2001). The Service Transport Layer uses information transport technologies in the same way as with supply-led models. However, the Service Technology layer describes the requirements of the system and the description, discovery, negotiation, delivery and composition mechanisms necessary for ultra-late binding of services, as and when they are needed. The Service Layer utilises this information to compose the system made from these services (Turner et al, 2003).

2.6.4 Components Off-the-Shelf

In COTS services,

> *"component middleware encapsulates specific services or sets of services to provide reusable building blocks that can be combined to develop enterprise applications"* (Gokhale, 2002).

COTS components are black-box, commercial components (see section 2.1.1), developed by teams specialising in COTS development and designed to reduce the amount of time and expense needed to compose software systems, which are offered in a competitive, global market (Torchiano et al, 2002). As explained by Gokhale (2002) and Stal (2002), components are commonly integrated via 'backbone' software architectures (the 'middleware') such as the Common Object Request Broker Architecture (CORBA), Sun's Java 2 Enterprise Edition (J2EE) architecture, and Microsoft's .NET and COM+ frameworks. Such architectures allow COTS components to be used independently of software platforms on which they are needed and languages in which they are written.

Amongst others, COTS exists to address the need for:

- Large-scale, complex enterprise systems that require the use of multiple software platforms and languages
- Quality of service (QoS) such as of efficiency, scalability, dependability and security

2.6.5 CLARiFi

Various systems aim to improve the environment in which system actors perform their roles. To take an example, the CLARiFi project at Keele University looked at the relationships between the actors in component-based software engineering, as shown in Figure 11. A *supplier* is responsible for creating components to integrate (assemble) into a software system. A *broker* stores component descriptions, which are used by an *integrator* for the purposes of component discovery, evaluation and selection (Brereton et al, 2000).

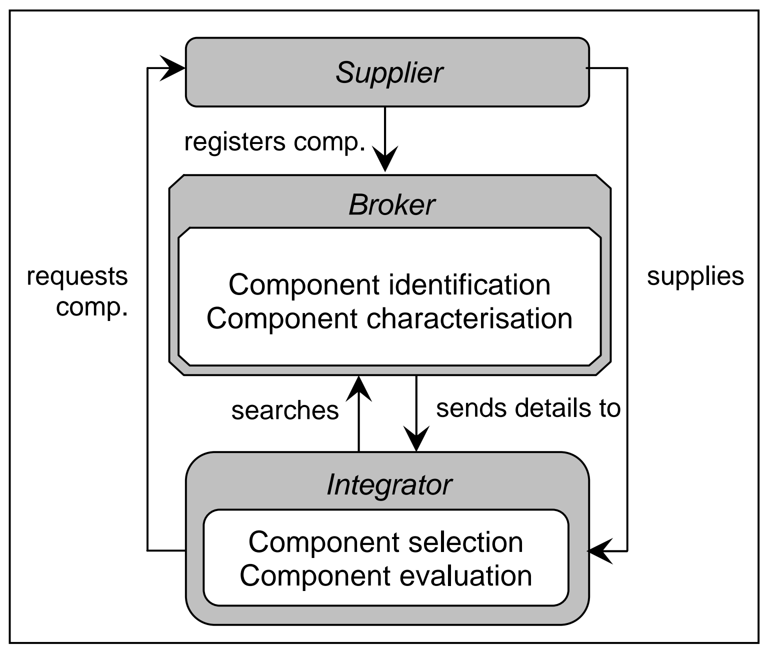
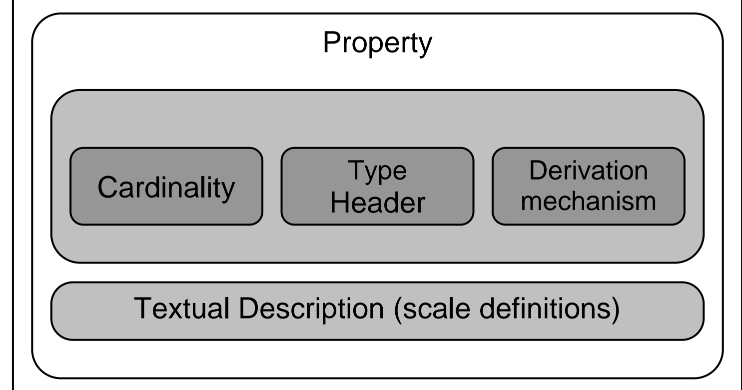


Figure 11 *The CLARiFi Component Relationship*

In the CLARiFi schema, each component has a set of properties, and this relationship is structured in the form shown in Figure 12 (taken from Brereton et al (2000)). The format is recursive, meaning that components may be composed of other components, and properties of other properties.

Figure 12 *The Details Inherent to a CLARiFi Property*

Each component has its respective set of properties, which take the form shown in the Figure 13. A type may be one or a group of properties. Whenever the type is a group, some form of definition of the scale (weighting) of each individual property accompanies Textual Description.



There are a number of types possible within the CLARiFi framework. A selection of these is shown in Figure 14 to illustrate how a description language syntax might be represented (Brereton, 2002; clarifi.eng.it). However, this document will not concentrate on the types to use in the process of formulating a description.

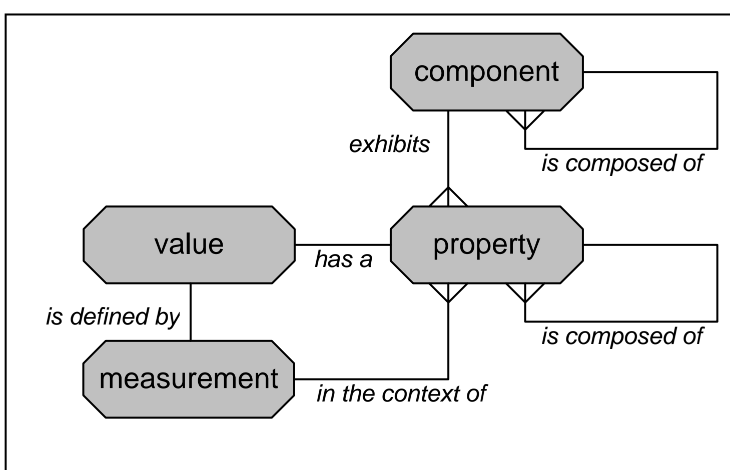


Figure 13 *The CLARiFi Property-Component Relationship*

Type	Example value(s)	Purpose
NOMINAL	Database, hospital, patient	An unordered value set
ORDINAL	Small, medium, large	An ordered value set
INTERVAL	-1.0:1.0	A scale with defined but arbitrary constraints
ABSOLUTE	0:10	As interval but with a base of absolute zero
RATIO	%	A count with definitive steps between points of the scale. No absolute base value necessary
DATE	22.02.02	A particular form of counting
SET	SET(functional abilities)	Used only with other types. Adjoins multiple types together under specified rules of use

Figure 14 *Example CLARiFi Properties*

3 Component Substitution Descriptions

Component selection requires some form of component description, so that a component's functionality is made known (Belletini et al, 2001; Bennett et al, 2001; Blair et al, 2001; Brereton et al, 2000; Damiani et al, 1997; D'Souza, 1998; Foster et al, 2002; Gomaa et al, 1999; Gonzàlez, 2000; Grundy et al, 2000; Heineman et al, 2001; Hendersen, 2001; Parrish et al, 2001; Saar, 2000; Saleh et al, 1998; Sousa et al, 2001; Sutcliffe, 2000; Talbert, 1998). Humans can select components for integration. However, with the ever-increasing demands for more complex interacting systems, this task is increasingly more daunting and costly (Rastofer et al, 2001). In addition, human error may occur at the integration stage. Consequently, the automation of this process is likely to be desired, particularly in large-scale systems and those that evolve rapidly (Damiani et al, 1997; Penix et al, 1997; Sutcliffe, 2000).

It was shown in the previous chapter that component descriptions can include:

- Functional properties - individual component functions, definitions, classifications and behaviours
- Non-functional properties – component roles, overall behaviour, constraints and dependencies

The conceptually simpler methods, such as simple keyword- and table-based descriptions, do not take into account non-functional properties that are difficult to quantify but useful for describing component functionality and usage more precisely and comprehensively. However, given that they are collections of property names, or types, and associated values, they provide the backbone of the newer descriptive methods, which do consider non-functional properties. It will be assumed that these methods, which include the non-functional descriptions described in the previous chapter, are more 'evolved' in the sense that they allow greater descriptive flexibility. For these reasons, they will be focused upon in the remainder of this document.

A component description also needs to address various other issues in order to be successful in a practical environment. Such issues include the degree with which the use of the description can be automated, the precision with which alternate components are searched for, component compatibility, the actors for which components are intended, and the nature and content of data that is necessary to be gathered and stored. Additionally, none of the descriptions introduced in chapter 2 allows complete automation in describing components and selecting appropriate candidate components for substitution. The indirect benefits of automated component selection are numerous, and would include:

- The elimination of human error into the description and selection process, assuming that the description language itself is not fatally flawed
- A decrease in the amount of time needed to describe components, assuming that the hardware and software being used is not so slow as to render a manual description process more time-effective
- A decrease in the cost of man-power due to the elimination of the need for manual work

The issues of automating component substitution will be considered in the remainder of this chapter. An explanation of the assumptions that will be made, and the focus of the remainder of this document, will then be set out.

3.1 The Issues of Precision and Complexity within Automation

It is possible to leave what and how to describe entirely up to individual component cataloguers, suppliers, composers and/or repository description procedures. However, this leaves a variety of different descriptions and contexts for integrators to search. Such descriptions can thus make the precision with which candidate components are found insufficient, because an amount of guesswork is likely in matching the descriptions given, and the contexts in which they are given, to the necessary search criteria. Significant arbitrariness and unreliability may thus apply to the components selected as a result of the unpredictable descriptions given, whilst the added work involved in interpreting property entries can make the system of component selection slower and less efficient. This work will therefore concentrate on the fuzzy matching descriptive methods in an attempt to make describing components as automatic as possible, with the nirvana being complete automation with no loss of descriptive precision. Having established the nature of description language to use, the next issue that will be discussed is the quantity of properties to use within the description.

Careful analysis of the descriptions required can minimise the amount of irrelevant and redundant data, therefore avoiding ‘bloating’ the description repository with unnecessary content and complexity and thus slowing down the system using the description (Gonzàlez, 2000; Sutcliffe, 2000). Furthermore, given a predictable, finite set of description categories, it is probably easier to check for inconsistencies and errors by singling out each entry and comparing it to sets of constraints allowed in each particular description.

A notable impediment to this approach might be the finite ways in which properties can be described to make searching quicker and easier. If fewer properties are provided, the description becomes more limited in its ability to describe precisely. Contrariwise, a greater quantity of properties would increase the range of descriptions to search for and check. It would also increase the number of descriptions that

the describer needs to know and the context in which it is to be used. One solution might be to compromise by having a large thesaurus of properties, in which the use of properties is strictly and exhaustively controlled, both in definition and context. The properties then need only be checked for consistency and errors once per description update, after which the search method can automatically match against the necessary criteria without worrying about errors.

Description syntaxes such as VDM and static behaviour sampling (see section 2.5.4.2) address the issues of descriptive flexibility and complexity, by systematically narrowing down candidate components until the desired number of candidates is reached. However, even these methods do not support the ultimate goal of complete, near-match automatic component selection. For example, static behaviour sampling requires manual insertion of test types and values. However, due to its advantages over the other description methods, which are described above, the static behaviour sampling method will be the foundational model upon which a new description language will be suggested that is capable of automatic component selection.

3.2 Exact- and Near-Match Components

It was shown in the previous chapter that description languages exist that can be used to match components exactly or “nearly” (approximately). In this document, an ‘exact’ match is considered to be an instance or copy of a component, or necessary component functionality, that is identical to the component or functionality intended for replacement. It should be inferred from this definition that a component description matching all of the keywords (and preferences, if any) of those in the query is also an exact match, assuming that the query is correct. A ‘near’ match is considered to be a component capable of replacing the previous one, either by modification or coincidence, independent of whether it is an instance or copy of the same component. This work will focus on support for both exact- and near matching, the former being implicit from the latter.

Near-matching components should operate to a satisfactory standard. By “satisfactory,” it is meant that the candidate near-match component should be able

An answer to this might be to have a key attribute pertaining to the domain of the component, so that a search method can identify the key attribute and look for the appropriate properties in response. This could nullify the need to provide properties, regardless of their relevance to specific components, because of their domain type. However, allowing an infinite amount of domain types could result in the same issues as with the argument for making *a priori* assumptions (see above).

The best solution might thus avoid the need for domain-specific properties entirely. However, in systems with many components, and in systems where large sets of descriptions are necessary, arranging components into domains can provide an effective means of reducing system complexity. The issue of using domain-specific components is large and not within the focus of automated component selection described here. It will thus be assumed that the components and/or their functionality described will be architecture independent, and that providing for component domains is not necessary.

3.4 Property Data Types

The issue of choosing appropriate data types occurs with the need to describe components through properties. In order to store, retrieve and interpret the values of properties, their types must be known.

Clearly, it is desirable to minimise the data types supported as much as possible without reducing the expressibility and flexibility necessary for the description, given that increasing the number of data types increases the complexity of the description language. A trade-off between higher flexibility and lower complexity is thus inherent with this issue. Furthermore, programming languages allow abstract data types and arrays as inputs and outputs, which will have to be supported in a real environment.

For the purposes of this work, abstract data types and arrays will be treated as other "primitive" types. As such, the management of data types will not be covered by this document. Furthermore, it will be assumed that the number of data types supported is thus not an issue.

3.5 Property Requirements

Section 3.1 identified static behaviour sampling as being an appropriate foundation upon which the NMACS description language should be based. However, static behaviour sampling itself does not allow the automation of gathering component property values, nor does it utilise test data for the purposes of locating appropriate candidate components. A minimal set of required component properties and their associated data types first needs to be identified that allows complete automation of the component description and search process in a domain-unspecific way. Minimising the quantity of properties reduces the complexity of the description and its use, as discussed in section 3.1.

It has already been shown that test data of inputs and outputs will be used in the selection of potential candidates. The issue remains regarding what input and output data is necessary in order to check the suitability of particular component functionality. There are various approaches to what constitutes input and output data necessary for NMACS, which can include one of the following:

- The complete range of exhaustive inputs and associated outputs, so that all possible inputs can be used to compare and check during the near-match selection process

- A set of non-exhaustive test inputs and associated outputs, so that sample inputs can be used to compare and check during the near-match selection process

- A set of test data of values at the extremes of the needed range, which can be generated purely from the input and output data types and ranges within the selection method

- As 3, except that the known exceptions can additionally be provided as test data

Clearly, the first method will work well with components whose possible inputs and outputs are severely limited, but with large components, the set of data will be extremely comprehensive, increasing the size of the component considerably. However, despite of this drawback, exact compatibility (if it is found) can be assured from this method.

The second method compromises the precision involved in testing the component for compatibility in order to reduce the amount of data stored with the component. However, the quality of the choices of which input values and associated output values to test is entirely up to the ability and prudence of the component's maker.

The third method requires no modification of or addition to the descriptions necessary for ACS. However, it may not be feasible or possible to test or predict all possible extreme data ranges. For example, a component may allow "-0" as well as "+0" to be returned, and the automatic test data generation method may not be aware of this exception (abnormal behaviour) to test for it.

The fourth method attempts to solve this problem by allowing the component maker to detail (only) any known exceptions with the component as test data to be used in the compatibility testing conducted using method 3. This limits the size of the component in a more controlled and less sporadic way than in the first and second methods. There is also the possibility that, in time, as new component exceptions are discovered through trial and error, additional data can be added. For these reasons, this last method will be accepted as being the best to use. Therefore, the input/output test data that will be gathered will represent both the extremes of each of the types of the component functionality, and also the exceptions to the usual input constraints.

The other properties that will be used in the proposed NMACS description, which are derived from the discussions that have been made, are:

- The component's identification, which should be unique to each component but which should be shared by different versions of the same component
- The component's version number
- The names of each of the component's functions
- The inputs and outputs of each function

A 'function', for these purposes, is the interface's access point to a particular functionality that the component is capable of providing. This set of properties is provided for the purposes of showing how the proposed NNACS description might be structured and used. Additional properties that might be preferential or necessary for finding an appropriate by integrators, such as component usage cost and performance ratings, will not be included in this exemplification. Instead, for simplicity and clarity, it is assumed that the system in which the NMACS description is to be used will be one in which additional properties such as these are not necessary.

4 Component Substitution Strategies

The suggested overall requirements for an automated, domain-unspecific description language have been identified. In the component searching process, the following stages will occur:

1. Static behaviour sampling, as described in chapter 2, will be utilised, using the properties described in section 3.5

2. If no appropriate components are established, the test data and exception properties will be utilised in order to find any candidates components that may exist by inserting inputs and exception properties and regarding those components that return results identical to the associated output values

The steps above outline the method necessary for NMACS. This chapter will continue by clarifying the processes that will occur in the utilisation of an NMACS description language. It will then examine the paradigms in which reuse occurs and the respective ways in which the NMACS description language might be implemented.

4.1 The Matching Process

As was explained in chapter 3, the process description language will use the same methodology as static behaviour matching, with the exception that it will not require the *manual* collection of test data results. Instead, the NMACS language will include an automated method for testing component compatibility, by comparing the input and output types, ranges and exceptions of both the previous and candidate components. Subsequently, the NMACS language will also have a completely automated property value collection mechanism.

4.2 Ways of Implementing the Description Language

Description languages contain multiple properties that describe the component to which they relate. Properties have particular data types and usually constraints upon the range of values that can be assigned to them. The following examines the structure of the properties that can be used for the new description language and the ways in which their values are gathered. The way in which these properties can be utilised to select candidate components with which to substitute will then be discussed.

4.2.1 The Property Value Gathering Process

Before component inputs and outputs can be compared for compatibility, the values first have to be obtained and stored in properties. These inputs will each have a data type and range, and possibly one or more exceptions.

- Data types can be obtained using the same method of signature matching as with static behaviour matching. It is necessary to know the data type of each input, since this information will affect the process of testing data type ranges. For example, a Boolean type requires only 'true' or 'false' to be tested, whilst an integer requires considerably more values that are whole numbers. In the event that one or more data types do not match exactly, data type range testing is important in the final decision of compatibility. If, for example, the candidate component's input is a 64-bit integer, and the to-be-replaced component's is a 32-bit integer, then the two components could be interchangeable even though that particular input value is not an exact match

- Test data from data type ranges can be obtained using a 'brute force' technique of applying each potential value in succession and monitoring the result. “Potential value” means any value that can

be accepted by the corresponding property of the component to be substituted. If all potential values correspond, then the range will be deemed appropriate

- If the result of the previous stage was an exception[25], then this fact can be recorded with the test data and utilised in the decision of whether to substitute with this component. Exceptions can further be obtained by passing the values that cause exceptions in the component to be substituted into the candidate component and seeing whether the result is identical

4.2.2 A Suggested Selection Process

The selection of a substitute component might be carried out under a range of policies, for example when:

- An exact match is to be found as soon as possible
- The "best" exact match is to be found
- An exact match is to be found given additional preferences
- The best near-match is to be found

Note that the terms "exact match" and "near match" as used in this document are defined in section 3.2.

[25] Here, "exception" refers to an exception to the rule, and not a Java Exception class or object.

Policy 1 (see Figure 15 and Figure 16) might be adopted for real-time systems where delay can mean out-of-date information, and would simply include using immediately and without further searching the first component found that matches the one previously being used. This may be termed **quick exact-match selection**, as this algorithm requires the least amount of time of the three.

Policies 2 and 3 (Figure 17 and Figure 18) are similar, where more time is available for component substitution. “Best” exact matches would be those components within the identical component list that are of the highest version number and which were updated and tested most recently and rigorously. As we are already searching for components given extra specifications, it is not too difficult to include additional preferences set by the integrator in the search. These may include such biases as the author, supplier and price. These two selection methods will be termed ‘**dynamic’ exact-match selection**, as the algorithm is sensitive to additional constraints (specifications, preferences) that the user may require in the component selection process.

Policy 4 (Figure 19 and Figure 20) would be invoked automatically in cases where an exact match was not found. In this case, the search would begin again using a different search method to match criteria *common* enough with another component to merit its use, even if it is not identical. This may be termed **near-match selection**.

The figures in this section suggest *modus operandi* with which the component substitution method would select components. They are given in informal pseudocode notation and diagrammatically. In short, for each component within each repository that may exist, if its description matches, it is used. This case occurs in the first stage of component substitution. If an exact match is not found, the second stage is invoked using near-match selection, in which a component is used whose description flags the component as being sufficient for component replacement.

Note: In Figure 15, Figure 17 and Figure 19, ← denotes “taken from” and → “stored to”.

4.2.2.1 Quick Exact-Match Selection

For each component within each repository that may exist, if the component identification, component version, function names and function and inputs and outputs match, then this component is used, the success reported to the system from which the algorithm was invoked, and the search is halted. If no such component is found and all components have been searched and evaluated, the lack of success is reported.

```
For each repository ← registry:
  For each component ← repository list:
    If ID, version, functions and I/O match
      Use component
      Return match found
      End search
    End if component name
  End for each component
End for each repository
Return no such match found
```

Figure 15 *Pseudocode Representation of Quick Exact-Match Selection*

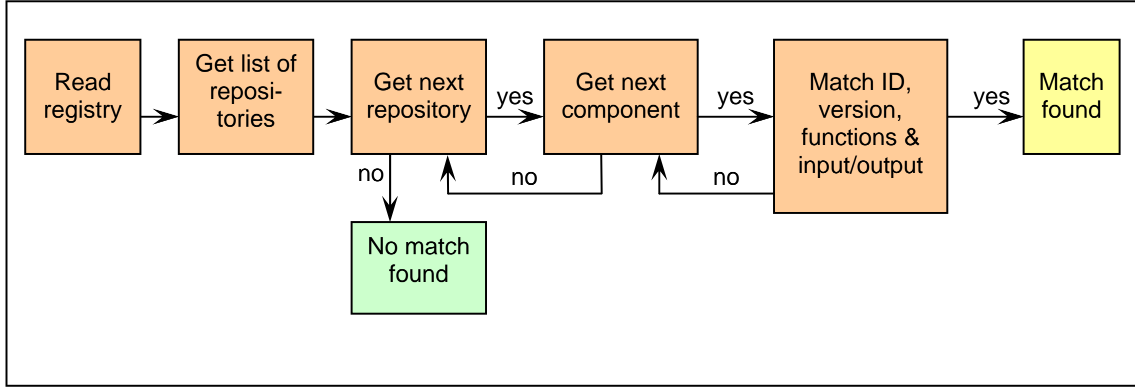


Figure 16 *Diagram Showing Quick Exact-Match Selection*

4.2.2.2 'Dynamic' Exact-Match Selection

With a list of components generated via a quick exact-match search, the component is used (if any) that complies best with the version number. The success or failure of this operation is reported to the place from which the algorithm was invoked.

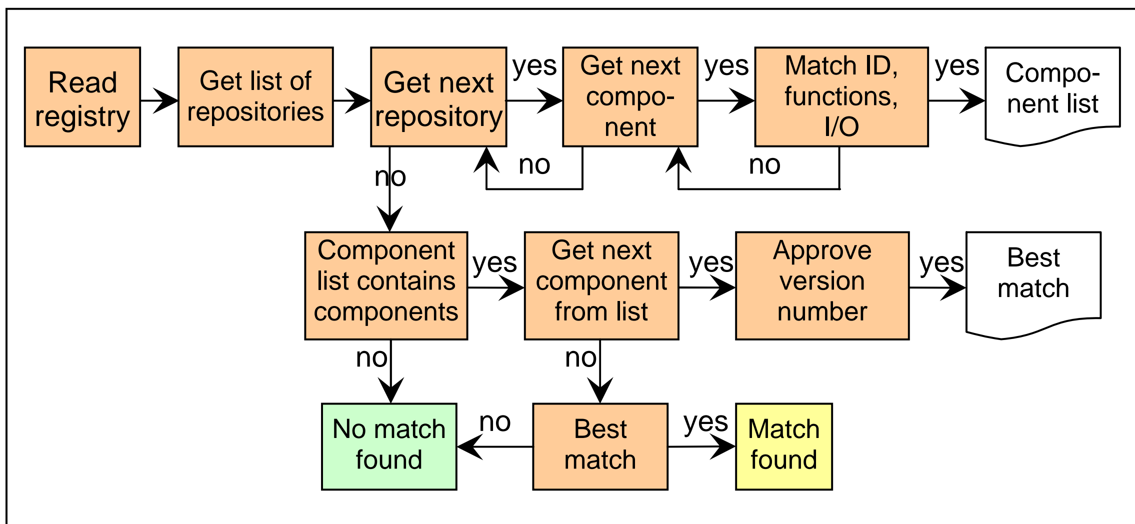


Figure 17 *Diagram Showing 'Dynamic' Exact-Match Selection*

```
For each repository ← registry:
  For each component ← repository list:
    If component ID, functions, input and output match
      Store component → component list
    End if component name
  End for each component
  For each component ← component list:
    If it has best version
        Choose component as best match
    End if
  End for each component
  If best match component exists
    Use best match component
    Return match found
  Else return match not found
  End if
End for each repository
```

Figure 18 *Pseudocode Representation of 'Dynamic' Exact-Match Selection*

4.2.2.3 Near-Match Selection

This algorithm requires the most extensive amount of code to perform its function, since there is little point searching for a near-match component straight away when an exact match may exist. Additionally, sufficient safeguards must be in place to ensure that the near-match is close enough to merit its use with the minimum of problems. First, a 'dynamic' exact-match selection process is initiated. If a component is identified for use at this stage, it is used, the success reported, and the search ended. If not, each method of each component located in each repository is checked. The types and ranges of input and output must match those of the previous component.

If this is so, the component is an applicable candidate for use. To test its integrity, an evaluation of the inputs and outputs is carried out. The test data supplied with the component previously being used should include various pairs of input types and values to test, and associated expected output results.

```
If component found ← dynamic exact-match selection
  Use component
  Return exact match found
  End search
End if component found
Else for each repository ← registry:
  For each component ← repository:
    For each function ← component:
      For each input-output pair ← method:
        If input type(s) and range(s) compatible
        And output type(s) and range(s) compatible
        And exception(s) compatible
          Use component
          Return nearest match found
          End search
        End if
      End for each input-output pair
    End for each function
  End for each component
End for each repository
Return nearest match not found
```

Figure 19 *Pseudocode Representation of Near-Match Selection*

Therefore, the new component should be able to accept these input data and give identical output, if it is to be appropriate. Note that the output data may be within an acceptable range, rather than an actual value, and that there may be multiple outputs. In the latter case, the output data should be encapsulated within an object.

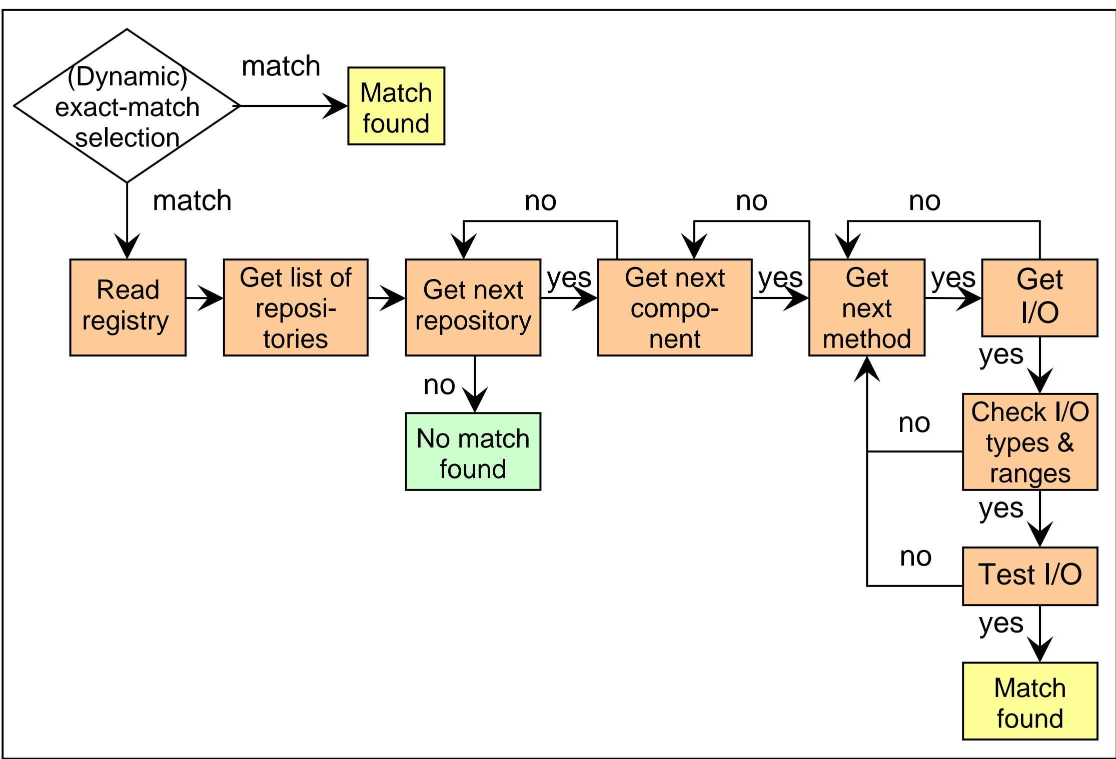


Figure 20 *Diagram Showing Near-Match Selection*

The test inputs of the previous component are used in a test-run of the new component on its own. If the given test outputs of the previous component match the actual outputs given as a result of running the new component, including the exceptions, then the new component is used. If not, the other methods of this component are checked, and so on for each component and each repository.

5 Evaluation Strategies

In the previous chapter, a component description and some strategies for using the description in order to support component substitution were proposed. However, upon identifying the possible use strategies, a number of additional requirements were shown to be necessary in order for the strategies to be facilitated. This chapter identifies what these additional NMACS requirements are and considers the steps necessary in establishing how Web technologies can be evaluated in their ability to facilitate NMACS. The Web technologies themselves are described and evaluated in the subsequent chapter.

Before a set of Web technologies can be identified and evaluated, an evaluation method has to be chosen and the overall goals that need to be met should be set out.

5.1 Choosing an Evaluation Method

It is necessary to evaluate each the criteria that are necessary for a candidate to support. A candidate whose functionality does not allow for these criteria will be deemed undesirable. A feature analysis is used precisely to determine these issues and form a structured assessment, and will therefore be used in finding an appropriate candidate(s) for the purposes of this document.

5.2 Explanation of Feature Analyses and Approaches

Although a feature analysis can be prepared in various ways, there are basic principles that make it an effective analysis method to use (Kitchenham, 1997). Feature analyses are useful for evaluating those features of technologies that have no objective value, such as quality, ease of use and usability, in order to come to a conclusion as to the suitability of the technologies being evaluated. It can also help to identify those features most important about particular technologies. This method of analysis incorporates an entire life cycle of experimentation, which includes (ibid., 1996 & 1997):

- Establishing the scope and basis upon which the evaluation is to take place
- Establishing the time, effort, material costs and assumptions that are to be used
- Establishing the features to evaluate and, optionally, a scoring mechanism for prioritisation
- Establishing an approach in which the feature analysis is to be applied
- Conducting the feature analysis
- Analysing the results

Feature analyses provide a flexible approach to evaluation. They are not limited by, and work well with a range of technologies, features to evaluate, levels of details and constraints of time and effort. Such approaches used to conduct feature analyses include (Kitchenham, 1997):

- **Screening Mode** – The use of technologies is not necessary in order to obtain evaluation results. Therefore relatively inexpensive, both materially and in terms of time usage. However, this means that such factors as scope and scalability are not directly evaluated. Performed by an individual, using only the technologies' documentation. Good for choosing from a large numbers of candidates. Can be used as the basis for more detailed analysis
- **Case Study** – Involves testing each technology according to set scenarios. Usually relatively inexpensive, but depends largely on the number of technologies evaluated and scenarios used. Only one person is required to conduct the study. Results depend heavily on quality of case studies chosen: Greater quality means that more resources are needed, but results can be interpreted with greater confidence. Can take a long time since technologies are used in order to obtain evaluation results. Good for evaluating scope, scalability, usability and extensibility

- **Formal Experiment** – Likely to be the most trustworthy and expensive. The evaluator plans the assessment criteria and procedures and conducts the assessment through separate, experimental subjects, who are not necessarily experts in the technologies assessed. The subjects score each feature with each candidate. Quality of results depends on quality of assessment criteria established and number of subjects. Difficult to produce experiment that reflects real-world scenarios and thus difficult to determine how well the results can be presumed precise on a larger scale. Additionally assumptions and lack of control of the knowledge, bias, experience, background and capability of the subjects might skew the results

- **Survey** – As with formal experiment, except the experimental subjects are knowledgeable in the use of the assessed technologies. The evaluator prepares and distributes a series of questions for the subjects to answer. Knowledge of real-life projects can thus be incorporated into the results, and less time is needed because knowledge about each technology is already there, and the results are as reliable as with formal experiment. However, as with formal experiment this brings with it the same problems of results skewing. Additionally, finding experts in all candidates, who can be relied upon to return their answers, may take considerable time and effort

5.3 Choosing a Feature Analysis Approach

The Web technologies that will be chosen will be evaluated according to their ability to support or implement the descriptions and the prerequisites desired for NMACS that were deemed necessary earlier. This will be assessed according to the technical (as opposed to suggestive, or corporate advertising) details given in their respective documentation. This limits the amount of time and effort needed. This significant issue would increase dramatically if the level of detail were to be extended to include physically testing each candidate. Therefore, the Screening Mode methodology will be chosen for its relative reliability in the face of the constraints on time and resources that are of concern in this

assessment. These limitations also highlight some of the limitations of this method of analysis, which will have to be taken into account. All scoring carried out will be subjective, drawing from an understanding of and being limited to the depth of detail of the relevant pieces of documentation.

Having established the aim of the analysis, the steps involved in the feature analysis process that are applicable here are:

1. Decide on a confidence level at which a technology will be regarded suitable or unsuitable
2. Establish requirements against which candidates will be evaluated
3. Select the candidates
4. Devise a scoring system
5. Carry out the scoring
6. Analyse the results

5.4 What to Evaluate

For ACS to be carried out, properties and their values need to be defined. Upon component substitution, they need to be available so that they can be used to evaluate the suitability of a candidate component. At this stage, the test data of the component to be substituted need to be accessed to apply to the candidate component, whilst the other properties of both the component to be replaced and the candidate component need to be compared. Therefore, an implicit requirement is the ability to store the properties and values of the component being replaced in such a way that these data are accessible even when the component itself is no longer available.

A component may fail and thus need to be substituted at any time during the execution of a distributed system. It is therefore important that the properties are accessible, and that components can be substituted, as and when needed. In addition, given that a number of suppliers may offer components, flexibility is required in choosing suppliers.

The following six features, against which each of the candidate Web technologies ("candidates") will be evaluated, address each of these issues in turn. The origin of the explanation of the need for a particular feature within this document, and a summary of the justification for its need, will be given. The order of the features relates to the NMACS properties, the NMACS strategy which uses these properties, and the suppliers of components respectively. Note also that the Web technologies themselves are all advertised as being able to facilitate the use of one or more description languages.

1. The candidate should facilitate the NMACS properties (given in section 3.5)

The suggested properties, and their appropriate values, need to be described in order for NMACS to be facilitated within the proposed selection strategies given in chapter 4.

Score	Explanation
0	It can neither facilitate nor be extended to facilitate the NMACS properties.
1	It can be extended to support these properties.
2	It already facilitates all of the NMACS properties.

2. Property values should be generated and stored automatically (see section 3.1)

In addition for the need for the NMACS properties to be described, their values need to be generated and stored automatically in order for ACS to be fully automatic.

Score	Explanation
0	Properties can either only be generated and written manually, or are not allowed at all.
1	Possible, but only after the candidate is extended to allow this to occur.
2	Values can be generated and written automatically.

3. Properties and property values should be ably stored as and when needed (see chapter 1, section 2.1)

Having demonstrated the ability to support the NMACS description, some way of storing the values generated should be provided whenever appropriate, in order for ACS to be fully automated. This would be necessary in cases where, during the search for candidate components, a component does not have an associated set of NMACS properties and values.

Score	Explanation
0	Not possible
1	Possible, but only after the candidate is extended to accept new properties and values whilst the system is running.
2	Properties and their values can be written automatically at any time during the running of the system.

4. Property values should be ably accessed as and when needed (see chapter 1, section 2.1)

In NMACS, appropriate components can be searched for after a component being used ceases to be available. This need may occur at any time during the running of the system. The candidate must therefore allow property values to be accessed at any time during this period.

Score	Explanation
0	Not possible, or possible only before the system is running.
1	Possible, but only after the candidate is extended to read values whilst the system is running.

2	Values can be read whilst the system is running.

5. Components should be substituted as and when needed (see chapter 1)

In addition to the need for properties and values to be generated and stored as needed during the candidate component search, a way of substituting components is needed once a candidate has been identified. This may happen at any stage during the running of the system.

Score	Explanation
0	Not possible, or the system must be stopped, modified and re-started before substitution.
1	Possible, but only after the candidate is extended.
2	This occurs automatically without the candidate having to be extended.

6. Component suppliers should be ably switched as and when needed (see section 2.1)

The origin of the component to substitute with may be different from that of the component to substitute. A way of switching component suppliers as and when needed is thus necessary.

Score	Explanation
0	The candidate does not allow switching of component suppliers.
1	Possible, but although only after the candidate is extended to allow this.
2	This occurs automatically.

6 Evaluation of Web Technologies

This section answers aim 5 as explained in section 1.7. The other aims are answered in chapter 7.

6.1 The Candidates

Having identified the features necessary for Web technologies to support in order to provide automatic component selection, a list of candidate Web technologies will be identified and outlined. Where applicable:

- The overviews of the selected Web technologies given conform to the most up-to-date versions of the documentation available at the time of writing
- Each explanation will take into account and mention explicitly any inherent bias that the source of the information given may have
- Only those details relevant to the goals specified will be mentioned

6.1.1 The Enterprise JavaBeans™ Specification v2.0[26]

The Enterprise JavaBeans™ Specification (EJB), the product of a number of organisations and fronted by Sun Microsystems, is a component framework aimed at supporting software vendors that specifically use and support the EJB framework for all components (DeMichiel et al, 2001) [27]. It is part of the Java 2 Enterprise Edition (J2EE) architecture, and defines a model for the development and deployment of reusable Java server components. It also extends the JavaBeans™ component model to support the construction, deployment and removal of server components in the distributed system (Thomas, 1998). Because of the duality of the term JavaBean, the component model that EJB extends are referred to as the *JavaBean* model and JavaBean components are referred to as *JBeans*, or *Beans*. A clarification of the distinction of these three models is given in Figure 21.

[26] The EJB 2.0 specification is as of writing in 'final draft' phase, and as such is subject to change. EJB 2.1 is already in its early draft stages, but for this reason will only mentioned here with a view to general enhancements to be expected, rather than any technical details.

[27] All information given in section 6.1.1 is taken from this source, unless otherwise stated.

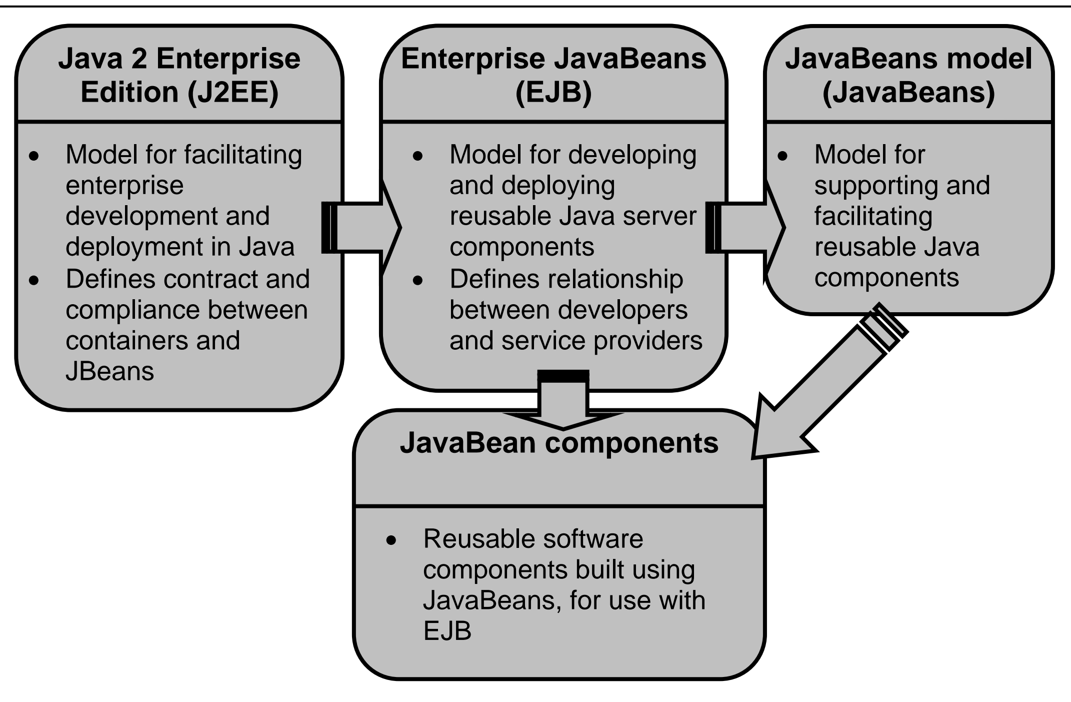


Figure 21 *Distinction between Three Java Distributed Architectures*

The component architecture, and thus the components themselves, used within EJB are called JavaBeans. Contracts are attached to the roles of each of the actors within EJB regarding their location, integration and deployment (see below bullets and Figure 22). However, it is accepted that one actor may take on more than one role simultaneously. Note that the term "actor" is not specifically defined within EJB, but can be considered an organisation or individual who has a role within the EJB framework as defined below. The word "entity" is given a specific usage context within EJB, so will be avoided when speaking generically.

The actors defined in the EJB model are:

- The **provider**, which describes how a JavaBean is structured and deployed, giving details of its dependencies

- The **assembler**, which uses this knowledge to provide instructions for composing the individual JavaBean, the environment variables and dependencies of which are provided in a set of types and values called a *deployment descriptor*. These details, which can be added and changed as and when needed basis at runtime if necessary, include:
  - The environment entry values and references to other values
  - A description of the fields
  - Abstract details of how to bind to other JavaBeans, including method permissions
  - Security roles

No implementation knowledge is possessed by either provider or assembler.

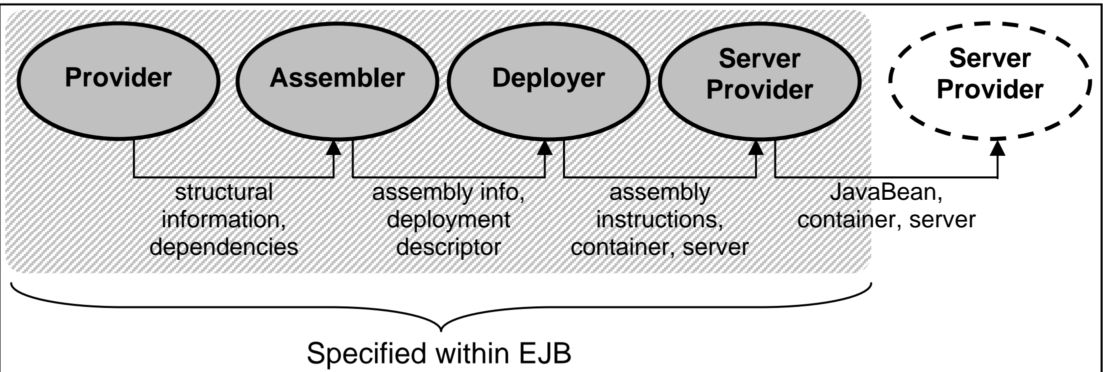


Figure 22 *A Simplification of the Relationship of, and Information Passed Between, EJB Roles*

- The **deployer** uses the assembly instructions to create and bind the JavaBean. It is also the deployer's role to provide:
  - A *container*, which details the constraints and dependencies upon which it is to be used, i.e. whether it is appropriate for a particular use and how it is to be implemented, within the context

of EJB. It also provides the application ontology and management and control services in an environment that allows components to be controlled and executed (Thomas, 1998; Nova, 1999)

- A *server*, from which it is to be discovered and retrieved

- The **server provider** provides the JavaBean (through its container) and the server. This actor is also responsible for ensuring an appropriate level of security, scalability, resources, versioning support, concurrency, system administration, transactions (see section 6.1.1.2) and other services

Although it is assumed that all components used within EJB are there primarily to facilitate JavaBeans that satisfy the EJB requirements, those that do not can also be supported if an appropriate container is built. Fragments of JavaBean code (at the assembler level) are shared amongst multiple clients. The self-contained blocks of code within an object are called *methods*. Remote clients must define:

- A **home interface**, which defines the methods that create and remove instances of this JavaBean and find other JavaBeans (see section 6.1.1.3)

- **Remote interfaces**, which ensure interoperability (see section 6.1.1.4)

- A **component interface**, which defines the methods that can be called from the software system in which it is to be integrated

- The object's **identity**, which must contain a primary key identifier and a method to compare it with other objects. Key identifiers are always unique objects, the only possible exception being separate instances – for example, versions – of the same object. The comparison method must thus be defined to test its relation with another object

The EJB architecture is platform independent, running via the Java Virtual Machine. Thus, any JBean component created on one platform can be used on another without modification. EJB also supports

any framework that uses the EJB architecture, making it to some extent language independent. However, given that EJB's primary aim is to facilitate components written in the Java programming language, significant work is necessary to map the functionality to EJB.

Furthermore, the specification is not exhaustive, and it can be difficult to guarantee that two clients will be compatible due to the vagueness and open-endedness of the EJB specification documentation. However, together with the few requirements necessary in JBean construction, this provides great scope for scalability and extensibility of JBean accessibility and operation, transaction and data handling, and specification enhancing (Nova, 1999).

6.1.1.1 JBeans

JBeans, the components of the EJB architecture, can be one of three types: Entity beans, session beans or message beans.

Entity Beans represent information stored in the database, i.e. components and properties. They are used in places where properties and descriptions apply to many entities, and are able to service connection and server failures. (There are two ways of representing the information through persistence – see section 6.1.1.5.)

Session Beans store details about the system. *Stateless* session beans represent business procedures, and do not carry information about the client using the bean. They are only used, and instances therefore only exist, for as long as the respective JBean is being used. They are relatively quick to produce, and are useful when information needs to be sent that does not change and is known *a priori*. *Stateful* session beans, on the other hand, carry information specific to a client in addition to business methods that changes with each transaction.

Message Beans enable JBeans to be communicated to different parts of the EJB system in particular orders, using the application server's queuing mechanism. Clients can put messages on the queue and message beans process these messages according to the communication rules that have been set. Messages processed via message beans can be distributed simultaneously with other types of beans. Additionally, message beans are stateless, meaning that they can send and receive messages concurrently.

6.1.1.2 Transactions

Within the EJB schema, the actor fulfilling the Server Provider role should provide transaction details. The Java Transaction API (JTA) supplies an interface for starting, joining and committing to transactions and performing 'rollback'. Rollback is the ability of a description architecture to store "time-slices" of parts of the distributed system at particular intervals. If part of the system fails, the system can revert to a former state at which it was working and continue from that point. A disadvantage is that information stored after that point will be lost, but the benefit is that the system gains a greater degree of fault tolerance. Transactions, described using the JTA, define the ways in which data sent between objects are to be treated. These definitions include:

- The requirements necessary in order to integrate a JavaBean (via its container)
- The JavaBean's dependencies on other JavaBeans (if any)

- Details of the JavaBean's functionality. Importance weightings are imposed using properties defined in a thesaurus, using one of the following rules[28] (Thomas, 2003; Nova, 1999):

    - BEAN MANAGED (*the JBean manages its own transaction context*)
    - NOT SUPPORTED (*the JBean cannot use a transaction context*)
    - SUPPORTS (*the JBean can execute with or without a transaction context)*
    - REQUIRED (*the JBean must execute within the transaction context*)
    - REQUIRES NEW (*the JBean must execute within a new transaction context*)
    - MANDATORY (the JBean must always execute within the client's transaction context)

6.1.1.3 The Home Interface and "Finding" (Searching for) JavaBeans

EJB ensures that methods, called *finder methods*, exist within each object inside the Home Interface. When these methods are automatically invoked with relevant parameters, it is the job of the method to find an appropriate replacement component, usually using the deployment descriptor provided in an *entity bean* (see section 6.1.1.3). It is thus, at the code level, the responsibility of the Provider to provide this functionality, and there are few rules for how this is conducted. At the abstract level,

[28] For clarity, the rules shown here have been given the 'informal' (non-technical) names for the transaction rules that the EJB specification supports. Precise technical details can be found in Sun Microsystems' *Enterprise JavaBeans Specification, v2.0*, Chapter 17.6.

possibilities include finding matches by primary key identifier, by interface and by particular functionality.

However, EJB v2.0 introduces a finder query syntax, which aids the finding and selection process. The semantics that encapsulates the syntax, and the syntax itself, is SQL-like, and can be compiled into SQL code for portability and optimisation. It can be used to identify abstract schemas, relationships, operations and expressions pertaining to the use of (part of) a JavaBean. The finder query syntax is defined in the deployment descriptor element, in the JavaBean's home interface.

The fact that information in an entity bean is lost when the bean is removed can present problems for finder methods. For example, finder methods search for JBeans within the database that match certain criteria. However, if these criteria, in the form of properties in the deployment descriptor object, are contained within an entity object, this information may not be available. The reason for this is that the object's database entry (and thus the information it was holding) will have been removed. A solution might be to use *container-managed persistence* (see section 6.1.1.5).

6.1.1.4 The Remote Interfaces and Interoperability

Interoperability is coordinated using the Common Object Request Broker Architecture (CORBA) and Java's Internet Inter-Orb Protocol (IIOP). This allows J2EE containers to communicate between J2EE-compliant servers on any platform, and addresses the issues of security by enforcing authorisation and access privileges to the data being transferred. The remove interfaces are written in Java, and the EJB enforces a specific structure that allows them to be mapped into Interface Definition Language (IDL) as described in the CORBA v2.3 General Inter-Orb Protocol (GIOP) specification.

6.1.1.5 Persistence

When a JBean is destroyed, its data is also deleted. Persistent (permanent) data can be stored within *entity beans*, which can be used by multiple clients. Entity beans usually exist for much longer periods of time than other types of beans, and represent underlying data to be used by other objects (Thomas, 2003; Nova, 1999).

A set of functions for managing persistence is described in the Home Interface. They can optionally be made to utilise the constraints set out in a deployment descriptor (see section 6.1.1.3) and include those for creation, removing, loading and storing methods and method data in JBean classes. These functions can be executed when objects are created, destroyed or loaded or removed from memory. This control can be passed to the JBean's container, which has the ability to intercept JBean calls and perform additional tasks before this event. This is termed c*ontainer-managed persistence*. Alternatively, storing such data within objects other than the container, such as entity beans, or session objects, which read and write directly from and to the database, is termed *session object persistence*.

6.1.1.6 Deployment Descriptors

Deployment Descriptor objects describe to the container the runtime preferences for managing and controlling its respective JBean. It can specify how a bean is to be created and maintained, as well as defining its name, interface names and environment properties by giving the necessary transaction semantics (Thomas, 2003).

The information in deployment descriptors is specified in XML format using any standard text editor, and is thus platform independent.

6.1.2 Web Services Description Language (WSDL) Specification v1.2

W3C's WSDL is a model for describing and locating components, or *Web services*, through sets of XML documents (Kontogiannis et al, 2003). It can be used to describe both abstract functionality and specific details of how and where functionality is offered. However, it has not yet officially become a W3C standard (Chinnici et al, 2003)[29].

The WSDL models for components and properties can be mapped into XML's infoset representation. Through this feature, WSDL information sent and received by Web services, or *messages*, can be bound (i.e. mapped, using a specific structure) onto a number of underlying protocols, such as SOAP,

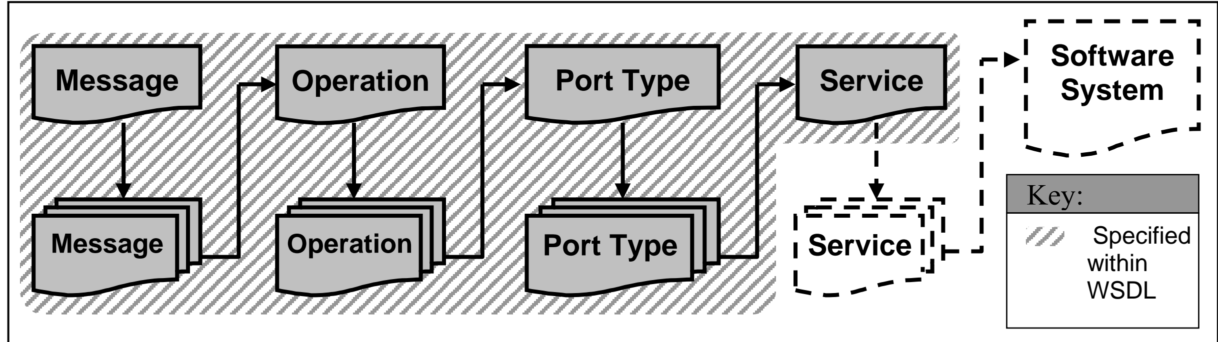


Figure 23 *Relationship between Levels of Abstraction in WSDL*

HTTP, MIME and DTD. Messages are structured using the XML property syntax and defined abstractly as a set of data type items and constraints (see section 6.1.2.1). Collections of messages,

[29] All information given in section 6.1.2 is taken from this source, unless otherwise stated.

which are contained within XML documents, are *operations.* These are grouped into *port types*, which are grouped into *services* (see Figure 23).

Messages predefine the data elements and functions in a way similar to programming languages such as C. Operations group together messages that access each other or are parts of a package. Port Types define Web services, functionality and message types (definitions), as well as how messages are to be transported, by the use of protocols, and effectively act as Web service component interfaces. They are protocol independent, meaning that they can be bound (mapped) to any underlying protocol capable of facilitating message passing, of which SOAP is one typical example. Once port types are bound to one or more protocols, the services can be deployed. Services can include links to other formal and informal documents, property types and functional description resources.

In addition to these four main types, attributes and constraints global to all items in a collection can be given and the grammar defined, as type and element pairs, via the XML schema.

Five major elements to the WSDL component model are based upon these levels of abstraction. Message operations, port types and services are constructed with reference to the definitions provided in these elements, in the same way that pages of a Microsoft Word document can be constructed using the master document or a template. They are as follows:

- **Definitions components**, which contain and define the structure for messages, port types, services and binding, including fault tolerance and definitions. Each definition comprises a collection of type definitions and value declarations

- **Definitions elements**, which provide the component name and corresponding XML namespace name, and the attribute information items to be defined by the definitions component

- **Attribute information items**, which describe items such as pointers to documentation and documents to include (section 6.1.2.2), types, and abstract message, port and service types

- **Port Type Operation components**, which describe part components by grouping messages into operations

- **Part components**, which are those parts of messages that are sent and received. Part components are termed *information element items* (IEIs) in XML. Such items may be name, input, output and error (fault) values

6.1.2.1 Constraints

In WSDL, all constraints must consist of a set of status keywords from an exhaustively defined thesaurus. These keywords include *must*, *must not*, *required* and *not required.* Although the way in which they are used is specific, there are no rules as to when and when not to use them. Consequently, defining constraints is versatile and open-ended.

6.1.2.2 Attribute Information Items (AIIs)

AIIs are defined in the XML Information Set. They include properties for name, value or reference to a value elsewhere, attribute type, and the owner element. AIIs are used within WSDL to encapsulate "concrete" (non-abstract) attributes, which can include:

- **Name** – the name of the component, element or item

- **Include** and **Import** – the component whose service definition should be included (inherited)

- The **type** of the element or item

- The **messages**, **operations, port types** and **service** to define

- The **binding** instructions

Extensions can be incorporated to the WSDL schema by defining additional declarations (to support protocols such as SOAP, HTTP and MIME) or Part Components (type and element pairs). These must use the definition constraints given in section 6.1.2.1.

A typical reason for extending the schema would be to provide a more specific binding protocol, in order, for example, to conform to a company's software policies or to allow the use of specific technologies. They include and import attributes that can be used so that they can be defined elsewhere in one document, to avoid unnecessary duplication of descriptions.

6.1.3 Distributed Component Object Model (DCOM) v1.0

Microsoft's DCOM, née Network OLE, is a distributed extension of COM and DCE. COM is a naming mechanism for instances of objects (components) and an extensible protocol that controls the state of objects dynamically, as needed at integration time and/or runtime, by manipulating instance names ("monikers") of these objects (Microsoft, 1996; 1997) [30]. It has existed and had a large market share for a relatively long time, and considered reliable. DCE is a widely used set of Web service technologies that address the issues of security, scalability, transparency (see section 6.1.3.3) and interoperability (OPEN, 1999).

One of the roles of DCOM is to provide a TCP/IP layer implementation for objects, and a wide range of operating systems is supported. Security, which plays a large part in DCOM, is extensible, customisable and configurable (see section 6.1.3.4), and built-in support, includes access and interception authentication, authorities and privileges. The authority of a user can be restricted to one of a range of levels of privileges through "impersonation," which can be set to one of anonymous (minimum privileges), identity (user must identify themselves or log in), impersonate (custom level) or delegate (maximum privileges).

Privileges, functions, properties and objects, transactions and components can be defined and constructed using the COM AppWizard, and the language used is very similar to C++. They are exposed via interfaces. Each "parameter" consists of a parameter name and one or more associated values. Parameter names are specified in a predetermined grammar, and additional ones can be included as necessary. Usually, values are Uniform Resource Indicators (URIs) that point to a distributed document in which the data is to be found. Such URIs can point to documents containing additional type, object and architecture constraints or data itself. An example of how pointers work within DCOM interfaces is shown in Figure 24.

---

[30] All information given in section 6.1.3 is taken from this source, unless otherwise stated.

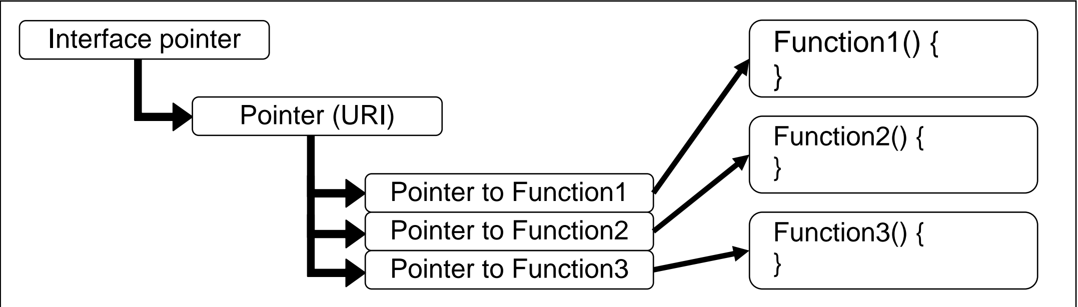


Figure 24 *Exemplification of the Use of DCOM Pointers (modified from [OPEN99])*

6.1.3.1 Versioning

The version is a standard parameter in a DCOM object. The content is loosely defined and not designed to be machine-readable. New versions of objects can be deployed at the time of integration and at runtime, by changing the URI of the parameter in the object pointing to the old object so that it points to the new one. This process is called *dynamic linking*. Additionally, new features can be put into new documents and their URIs added to the list of URI values in the relevant parameter of the relevant object. This means that features are *inherited*, in the object-oriented sense of the word.

Different versions and instances of objects may be given the same name and identifier. This is not usually a problem: If a component instance is capable of particular functionality then the name of the instance being used is largely irrelevant. However, the way in which an instance performs a function may change, and this may be important in some systems. The use of *monikers* averts this problem, where instances are given unique identifiers that can be accessed when necessary. Monikers are bound to their respective objects and defined in their own interfaces.

6.1.3.2 Interfaces and Modifying or Searching for Components

When a component's definition changes, the interfaces are either substituted completely or extended with additional interfaces to reflect the changes, as opposed to altering existing interfaces. The reason

for this is to allow querying of functionality whenever it is needed. Interfaces can be extended in two ways:

- *Containment*, or *Delegation* – the previous interface is 'wrapped' inside the new one, so that its details can only be accessed by first accessing the outer interface, which then grants or denies permission to certain details

- *Aggregation* – the new interface gives a reference to the previous one, so the object accessing the interface has direct access to all details in all respective interfaces

Thus, by averting the availability of obsolete interfaces, which are removed completely, and by accessing interfaces "just in time" rather than holding instances (copies), some of the versioning problems surrounding, for example, Microsoft Dynamic Link Libraries (DLLs). Furthermore, the problem of the possibility for interfaces with different versions and identical identifiers to exist is avoided.

An object can dynamically and abstractly request an object (optionally of a specific version) that has specific features by calling a *dynamic function query*. Such queries have two roles:

- To locate objects that provide the necessary features

- To ensure that these features are provided with the appropriate levels of access (see the details on "impersonation", chapter 6.1.3)

The querying process is implemented by an interface called QueryInterface, and defined using Microsoft's comprehensive and flexible Interface Definition Language (IDL). The definition consists of a series of properties and function methods, together with input and output types. The code for the query itself is written in the Java programming language.

6.1.3.3 Transparency

Objects can be deployed and redeployed dynamically at runtime, and this is dealt with in DCOM at a level of abstraction above the platform(s) upon which it is being used. Various levels of abstraction are supported, including TCP/IP, UDP, IPX/SPX and NetBIOS. CORBA is not used within the standard DCOM architecture, and properties are inherited from DCE. However, support for the CORBA model of describing components and interfaces is provided. Additionally, any other platform-neutral frameworks can be incorporated into the system that in turn incorporates DCOM.

DCOM is also location independent – meaning that it is easily scalable, language neutral – meaning that it can be used to integrate components written in a wide variety of programming languages, and runs on Windows, Macintosh and Unix platforms. However, implementations on non-Windows platforms are new and their reliability and compatibility is yet to be seen over the long-term.

6.1.3.4 Extensibility

DCOM uses Microsoft’s Interface Definition Language (IDL) for specifying additional constraints and parameters. Microsoft IDL is an interpretation of OMG’s IDL, which maps data and constraints onto specific programming languages. Therefore:

1. DCOM can in theory incorporate objects written in any IDL-tolerant programming language

2. DCOM objects, constraints and parameters are written in any IDL-tolerant programming language

DCOM's IDispatch function has a number of useful roles. Firstly, it allows object properties and property data types to be read dynamically, via “get” functions. Secondly, it enables objects to be tested with a given set of arguments, via the “Invoke” function. There are additional functions (OPEN, 1999).

There is a minimum level of fault tolerance as standard in the DCOM architecture, which involves pinging the object instances in the system at arbitrary intervals and failing after a time-out period. In the case where information is lost due to losing the use of an object instance, then a “fail-over” system is resorted to. In fail-over, the system object that located and bound the component is called again, and it is then the job of this system object to allocate a new object in its place. DCOM then reinstates the system from that point automatically.

As far as security goes, DCOM takes advantage of standard Internet certification and authentication techniques and the built-in security functions of Windows NT, when applicable. This is done via access control lists, which specify which users may be granted access to which services, components and component data and functionality. It is also configurable to implement customised levels of access privileges and security measures, using the DCOM Configuration Tool. A simple example of the built-in DCOM security architecture is given in Figure 25 (interpreted from a figure given in the main source).

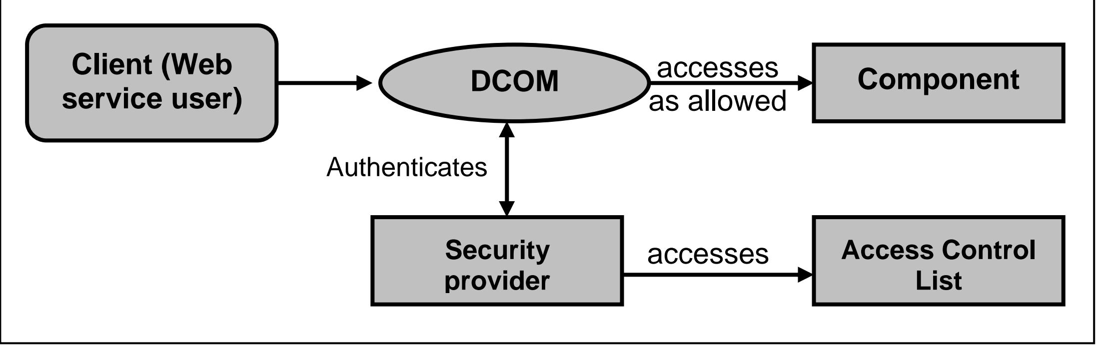


Figure 25 *Standard DCOM “Security by Configuration” Architecture*

6.2 Scoring System

A scoring system needs to be assigned for each of the goals established. Although a simple ‘yes’ or ‘no’ may suffice, an additional merit will be given to those technologies that, in addition to providing the *means* to support, actually *implement* a particular function. Thus, there are three possibilities: ‘No’, ‘Yes – extensible’ and ‘Yes - built-in’. These will be given scores of 0, 1 and 2 respectively for each of the required functions.

Given that each feature to be scored is considered mandatory, it is unnecessary to determine whether one feature is more desirable than another in order to prioritise (regard as being more important) one feature over another. However, a score of at least one for each question is necessary in a suitable technology.

6.3 The Scoring

Having discerned the evaluation method to use, the candidates to evaluate and the ways in which the evaluation method will be used, it is possible to conduct the evaluation on each of the candidates in turn. A discussion of the results, using the scoring system identified in section 6.2, will be carried out in chapter 8.

6.3.1 The candidate should facilitate the NMACS properties

Technology	Score	Details
EJB	1	Events and property types have to be defined and stored as persistent data elements in the database via entity beans. Definitions are made using home and remote interfaces using an XML-based deployment descriptor and Java code. Once the property types have been defined, the developer can use the resulting deployment descriptor to create properties in Java code manually, to be stored in JBeans. (Nova, 1999)
WSDL	1	WSDL 'definitions components' define property types, message types and document pointers, called 'attribute information items'. Data types & constraints are composed using the XML property syntax in Messages, whilst functionality and message types are defined in Port Types. Data elements and functions can be defined manually in 'part components' inside Messages, again in XML format. Each element holds a WSDL and XML namespace name and a set of 'information element items,' which can consist of input, output and error (fault) values.
DCOM	1	Constraints and types can be specified in IDL, in a specialised document, and then mapped onto specific programming languages. DCOM parameters consist of a parameter name and one or more associated values. Although a descriptions and constraints list does not exist, one can be specified manually according to the constraints specified using IDL, and are variable according to the specific needs of the component vendor.

6.3.2 Property values should be generated and written automatically

Technology	Score	Details
EJB	2	Properties are stored inside and provided by the JBean's container. Clients or JBeans cannot access containers themselves. However, they can be made to intercept JBean invocations and transactions, and provide additional functionality when necessary. This functionality includes writing information to properties. Given that EJB uses Java to implement the code to do this, abstract data types can be created to store constraints on and ranges of values, as well as the values themselves. (Nova, 1999). This can then be used to automate the process of writing property values thereafter.
WSDL	2	WSDL is an interoperable language that supports multiple protocols. It is possible to use or extend one of those protocols that allows types, constraints, ranges and values to be written to properties by "inspecting" components directly. After this initial process, writing property values can be automated. Although WSDL does not support the automatic generation and writing of values, it supports protocols that do, in effect giving the same results.
DCOM	2	The information regarding data types, constraints and ranges is hidden and not directly accessible within the DCOM architecture. However, a parameter can be used to point to a specialised document containing this information if and only if the component producer knows and enters this information before integration. This can be automated via IDL code.

6.3.3 Property values should be ably written as and when needed

Technology	Score	Details
EJB	2	Values can be written by Client and/or JBean messages at any time during the execution of a distributed system through the component code.
WSDL	2	Properties can be written at any time during the execution of a distributed system, assuming that the protocol that has been used supports these actions. (However, given the flexibility with which WSDL allows such actions, considerable effort is required to implement this process.)
DCOM	2	New versions of components can be integrated by changing the URI of a document to point to the new one. An additional document containing the new values must be created, and the URI must be changed to point to this new document.

6.3.4 Property values should be ably accessed as and when needed

Technology	Score	Details
EJB	2	Once the values have been written, they can be accessed by Client and/or JBean messages at any time during the execution of a distributed system. Additionally, finder methods, which belong to the Home Interface, can be modified to change property values.
WSDL	2	Properties can be inspected at any time during the execution of a distributed system, assuming that the protocol that has been used supports these actions.

DCOM	2	This is done via the document's URI.

6.3.5 Components should be substituted as and when needed

Technology	Score	Details
EJB	2	Via the respective JBean's search method. Other JBeans can only be selected from those advertised on the database. Different versions of JBeans are treated as separate components, so confusion over different versions is likely to be minimised. The process of substituting components is controlled directly by the EJB infrastructure and implemented via Java.
WSDL	2	Components can be replaced by redefining messages (data elements and functions) in new 'parts components' deploying the new object by altering the pointers in the relevant information element items to point to the new schema.
DCOM	1	Components can be replaced by changing the URI (pointer) of the relevant document to point to the new one. However, compatibility must be ensured *manually* beforehand or the system will fail.

6.3.6 Component suppliers should be ably switched as and when needed

Technology	Score	Details
EJB	1	Yes, but redeployment of JBeans from "vendors" (suppliers) is not specified in the database, and has to be

		done manually.
WSDL	2	Yes, by defining a new 'service' schema and deploying this new document by altering the pointers in the relevant information element items to point to the new schema.
DCOM	2	Yes, again by changing the URI of the relevant document to point to the new one. DCOM URIs can be web addresses, thereby resulting in vendor independence.

6.3.7 Total Scores

Technology	Score	Details
EJB	10/12	Facilitation of NMACS is possible via search methods. However, the locating of relevant components may be hindered by the limitation of component information available in the database.
WSDL	11/12	Facilitation of NMACS is possible by altering Messages, Port Types and Service objects and changing the respective attribute information items.
DCOM	10/12	Facilitation of NMACS is possible by creating new documents with the relevant information and changing the relevant documents' URIs to point to the new components and respective sources of information. However, additional preferences unique to particular integrators can only be pre-set, and thereafter not altered.

7 Evaluation of (NM)ACS – Case Study

This chapter describes a case study that can be used to evaluate the suggested (NM)ACS component selection process. The aim of this chapter is to show how (NM)ACS can be used in a software system by identifying an example, and to use the example in order to evaluate whether the (NM)ACS method is successful in achieving the goals explained in chapter 1.7.

7.1 Overview

The case study makes a GUI wizard available to end users. Its purpose is to allow a user to make choices in organising a train journey, and to return the price of that journey. The underlying system happens to be Java-based, and the GUI is designed and constructed using the Java™ Swing® package. The Swing package is a set of visual components that can be used to construct a GUI. It makes available a range of well-known components, such as text fields, check-boxes and text labels, which will be familiar to users of virtually all GUI-based operating systems.

The available storage space of the GUI wizard system is limited. As a consequence of this, only some of the basic standard Swing components are used. Additionally, the options available on the GUI (and so the GUI's appearance) can vary, according to the user's authority and the details entered by the user. Thus, although the GUI is constructed and displayed locally, using the system state, the business logic involved in deciding the GUI's appearance and state is accessed remotely.

Due to the fact that similar GUIs will be used amongst train operators who use similar GUI software, the repository of available components is shared. However, the GUI wizards may vary from operator to operator.

When the need arises to use components outside the GUI wizard's local system, the following steps need to be taken (see also Figure 26):

1. The GUI wizard accesses a middleware component, passing it the new user selection(s)

2. The middleware attempts to access the pre-specified component. (This component's name and details would be specified at the system's design stage). If it is available for use, then it can be used (skip to point 5, below). Otherwise:

3. The middleware decides upon the functional details (description) of the component necessary to perform the task, including test data

4. This description is used by (NM)ACS in order to search for a component that is capable of carrying out the task. An appropriate component is chosen

5. The middleware makes use of the chosen component, and updates the system's state

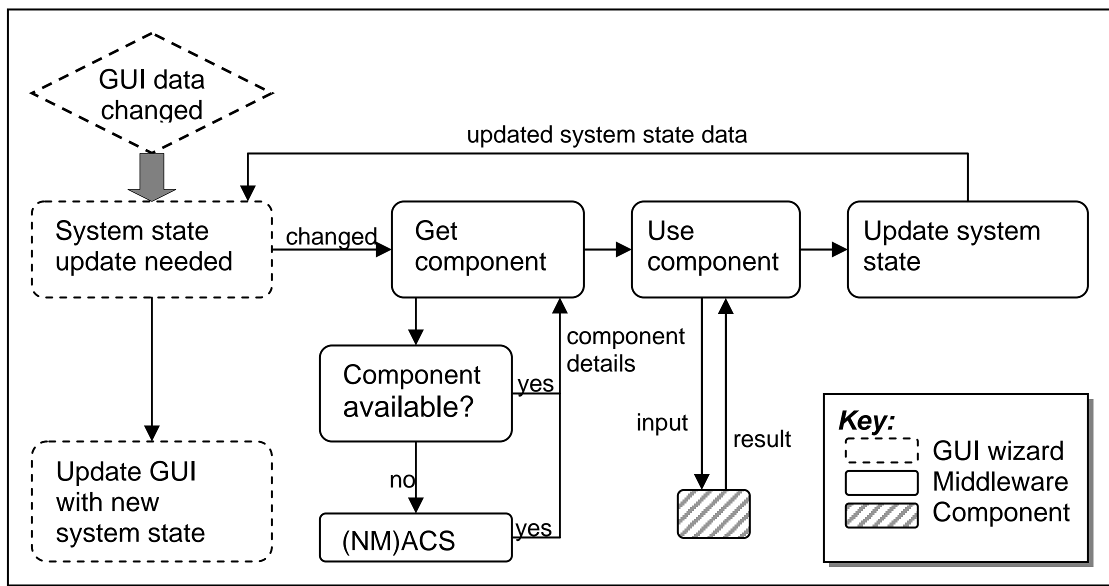


Figure 26 *Diagram to show flow of information through the case study*

The GUI wizard then updates the GUI based on the new system state.

The case study demonstrates how (NM)ACS can be used to drive a GUI whose appearance and state are utilised locally and decided upon remotely[31]. The same repository of components is used by multiple train operators, which means that there is the possibility of exact component matches. However, the variability of GUI wizard designs amongst the operators means that the intended use of some components may also vary. The functions of these components may be adequate enough to be considered a viable near-match alternative.

7.2 The Swing Components Used in the Wizard GUI

Figure 27 lists the six Swing components ("widgets") used in this case study, together with descriptions and the corresponding symbols that this work will use to represent them.

Widget	Description	Example
Text field	Allows entry of one line of text	
Combobox	Allows selection of items from a predefined and finite list	Abc ▽
Radio button	Allows one item out of a group of items to be selected	◎
Checkbox	Allows each item to be selected or left unselected, from a group of items	✓
Label	Displays predefined text	Abc
Button	Performs a task	Abc

Figure 27 *Diagram to show flow of information through the case study*

[31] That is, the decisions to make regarding the GUI's state are made remotely via remote access to components in the repository. The updated state is then used by the local system to drive the GUI.

The user can interact with the GUI by making selections and sometimes by entering text, depending on the widgets made available. As a consequence of the user's details[32], the GUI can respond by:

- Making widgets active (the user can enter data/make a selection) and inactive (the user cannot enter data/make a selection)
- Validating data entered, and choices made, by the user
- Displaying messages, for example to show errors and ask for confirmation

Note that if a widget is *inactive*, then it is displayed but cannot be used (i.e. is 'ghosted out'). If a widget is *active*, then it is displayed and can be used

7.3 Overview of Wizard GUI's Structure

In the remainder of this chapter, a 'page' refers to a collection of on-screen widgets, which represent one step in the user's progress through the wizard exemplified in this case study. One page thus contains all of the widgets (and thus potential choices) for one 'step' through the wizard. The user will typically make a number of selections and enter data on the current page. If the details are considered valid, then the wizard will progress to the next page. Otherwise, an error message will be displayed and the user will have to amend the page's details before progressing.

[32] To reiterate, the representation of the GUI is dependent upon the identity of the user, as well as the data and selections that the user enters and makes.

The appearance of the next page always depends on the state of the system, which changes depending on the selections made by the user. The system state, in this context, is thus deemed to be the information elicited from all of the choices (data and selections) made by the user so far. Additionally, an individual widget's appearance may alter, depending on choices made elsewhere on the same page. The page state, in this context, is deemed to be the information elicited from the choices that the user has made on the given page at any time. The page state can determine whether a widget on that page is active or inactive. To reiterate, state is *elicited* from the choices that the user makes, which can either be direct (the choices themselves) or indirect (inferred information, such as whether to enable or disable a widget).

This case study will take the form of a 'wizard' - a sequence of related pages that, together, allow the user to perform an overall action. The wizard asks for the user to make choices in organising a train journey, and returns the price of the journey based on these details.

7.3.1 The Design Rules

There are three important factors inherent in the design of the train operator system:

- The **GUI rules**, which determine the GUI's appearance, *use* the information that the GUI has direct access to - the system state
- The **business rules** are used to *decide* the system state, and are known by remote components
- The **state rules** are used to *maintain* the system state. It is the job of the middleware to accept update requests from the GUI, and to use remote components to keep the system state up-to-date

Data validation is considered part of the business rules, because it involves knowledge necessary to alter the system state. However, the process of displaying error messages is part of the GUI rules, because it is an intrinsic part of the GUI's appearance.

The GUI has knowledge of the selections the user makes and how to use the middleware, but has no knowledge of external components. The following GUI design rules apply, which are known locally (i.e. by the GUI system):

1. By default, a widget's state is active
2. By default, a widget's value or selection state is cleared
3. If a widget is made inactive, then its value or selection is cleared
4. The appearance of the GUI depends on the system and page state
5. State is read only
6. If some data on the page are not valid when the "OK" button is clicked, then an error message is displayed

The middleware knows how to use components in the repository, based on information passed in by the GUI system. However, it does not know what data the user enters into the wizard, nor does it know how the external components make their decisions. The following middleware design rules apply:

1. A state attribute can be data entered by the user, a choice made by the user or a decision made as an indirect result of another state attribute
2. The system state includes the name, value and enabled/disabled status of each attribute

3. The system state should be brought up-to-date whenever information is received from the GUI

4. If data is found not to be valid, then this fact should be recorded in the system state, along with an error message relating to the error

External components know how to make decisions based upon the data supplied by the middleware. However, they are not aware of the wizard or the system state. The following business rules apply, which are decided by components:

1. Users do not have to register in order to use the wizard

2. Only registered members may book business-class tickets

3. Only students and OAPs may buy economy tickets. These details are entered on registration

4. The user may (only) travel to and from the pre-defined list of stations

5. If a return ticket is requested, then the user may only return to the original place of departure

The predefined list of stations for all train operators, and the price of a standard single ticket between any two of these stations, are shown in Figure 28:

	Stoke	Stafford	Blythe Bridge
Stoke		£8	£5.50
Stafford	£8		£10
Blythe Bridge	£5.50	£10	

Figure 28 *Train Stations and Ticket Prices*

7.3.2 Operation of the GUI Wizard

The wizard needs the following details, before the ticket price can be displayed:

- The user's details
- The departure and destination stations
- Whether it's a single or return journey
- The class of travel

It is assumed that user details are always available to external components. The following is an overview of each of the pages in the train journey pricing wizard:

- **Page 1** – Ask whether the user is registered. If so, ask for user name and password
- **Page 2** – Ask for the place of departure, the destination and whether it is a return trip
- **Page 3** – Ask to select a ticket type (economy class, standard class, business class)
- **Page 4** – Show the ticket summary and price

Each page contains two buttons, situated at the bottom-left-hand corner of the page:

- 'Back'. This displays the previous page. If this is the first page, then 'Back' is inactive
- 'Ok'. This validates the details. If details are invalid (i.e. the validation failed), then an error message is displayed. Otherwise, the next page is displayed. If this is the last page, then the first page is displayed – i.e. the user is taken back to the start of the wizard

7.3.3 Error Messages

Error messages are GUI windows, which inform the user of a detail that must be amended before continuing to the next page. They contain:

- The details of the first error that was found
- A ‘Back’ button that returns to the page that generated the error

7.4 The GUI Wizard’s Appearance

The wizard uses error message windows and pages that represent steps in the progress of the wizard. This part describes the appearance these two concepts on the screen.

7.4.1 The Appearance of Error Messages

If, at any stage of progression through the wizard, the user makes an invalid selection (i.e. the state of a page is incorrect), then an error message is displayed. Figure 29 shows an example of such an error message requester.

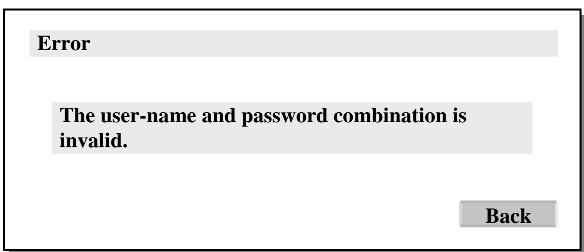


Figure 29 *An example of an error message.*

7.4.2 Page 1: Ask Whether the User is Registered

A checkbox asks whether the user is registered. If checked (i.e. ticked, so that the user chooses ‘yes’), then user-name and password fields become active and the user must enter user-name and password.

‘Back’ is always inactive. ‘Ok’ validates the user-name and password combination (if appropriate), and goes to the next page if the details are valid.

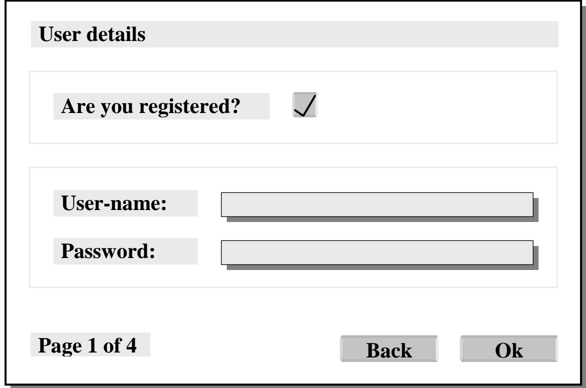


Figure 30 *Page 1, The “User Details” Screen*

7.4.3 Page 2: Ask for Departure, Destination and Whether it is a Return Trip

The “Depart from” and “To destination” combo-boxes list the stations to choose from for the departure and destination stations. One must be selected for each. Additionally, the user may request a return ticket by checking the checkbox.

‘Ok’ validates that a departure and destination station have been selected and that they are not the same, and goes to the next page if the details are valid.

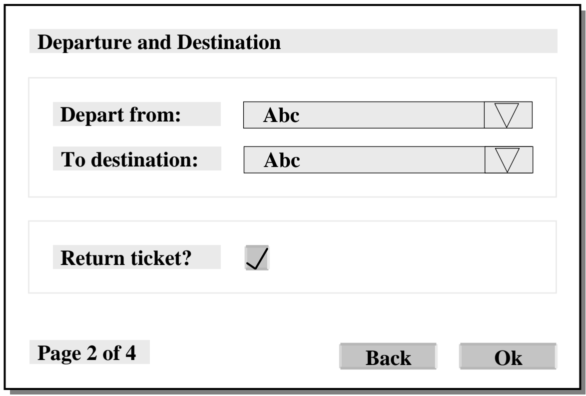


Figure 31 *Page 2, the “Departure and Destination” Screen*

7.4.4 Page 3: Ask to Select a Ticket Type

A series of radio buttons allow the user to select an economy class, a standard class or a business class train ticket. If the user is not registered, then the economy class and business class radio buttons are inactive. Otherwise, if the user is registered but not as a student or OAP, then the economy class radio button is inactive.

‘Ok’ validates that a ticket type has been selected, and goes to the next page if the details are valid.

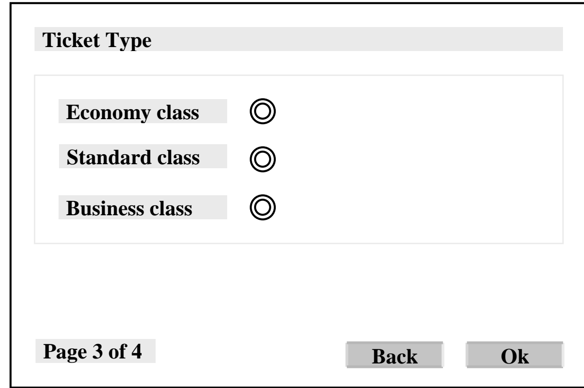


Figure 32 *Page 3, the “Ticket Type” Screen*

7.4.5 Page 4: Show Ticket Summary and Price

This page displays a summary of the ticket chosen, and its corresponding price. All of the widgets are always inactive, except for the 'Back' and 'Ok' buttons.

If no user-name was entered, then the 'User-name' field contains the text '(None given)'. The "Ticket type" text field contains the class of ticket ("Economy class", "Standard class" or "Business class", depending on the selection made on page 3), and whether the user selected a single or return trip.

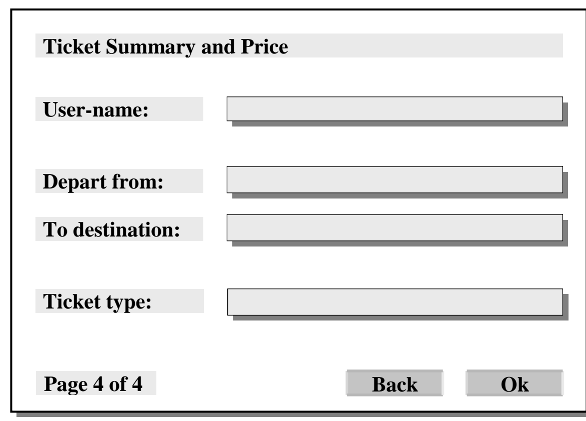


Figure 33 *Page 4, the "Ticket Summary and Price" Screen*

'Ok' resets all of the details entered by the user and returns to the first page of the wizard.

7.5 Technical Details concerning GUI Construction

On the local system, a page is constructed given a state. The state is the set of values for a group of variables, such as the system state (all the variables in the GUI) or a page's state (all the variables on a particular page of the wizard). A variable's value is altered depending on user input. For example, one variable is a Boolean that stores whether the user is registered, whilst another is a string of text containing nothing (user not registered) or the user's password. Each widget is 'bound' to one or more state variables applicable to it. A widget that is 'bound' to a variable changes its visibility, enabled/disabled status and/or content depending on that variable.

The following section gives technical details about how the values are stored, used and decided upon, in order to organise the GUI's state and appearance.

7.5.1 Constructing Pages

Each page is constructed (or refreshed, if it has been visited before) depending on the system state, unless 'Back' is selected. The state of the next page (e.g. widgets display and whether they are active or inactive) depends on the state of the previous page. The 'Back' button thus only has to go back and redisplay the previous page, whose details are still as they were when the user last entered them. It is not necessary to reconstruct a page when 'Back' is selected, because that page's state is already known and has not changed.

The middleware is the link between the repository and the GUI system. Its job is to use external components, using (NM)ACS if necessary, and to update the system state. The relationship between these three processes is shown in Figure 34.

The local GUI uses internal functions to build and maintain the GUI using the GUI's state, and external functions to control and verify the GUI's state, depending on user interaction with the GUI. External

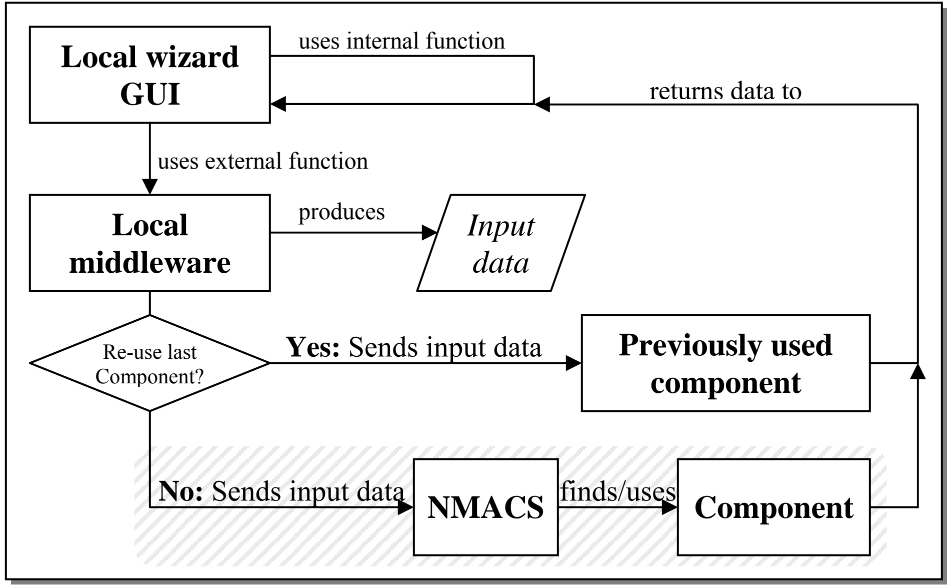


Figure 34 *Diagram showing the Relationship Between Parts of the Wizard System*

functions are utilised indirectly through the middleware.

The middleware elicits the input data that would be needed by a remote component in order to decide a new state, find a component able to process those data, and update the system state accordingly. If the system state is in error – that is, the user made an inappropriate decision or entered incorrect data – then an error message is stored which is used by the GUI in order to prompt the user to correct their decision or re-enter data.

If a previously used component is still available that can perform the necessary functionality, then the same one is reused with this new input data. Otherwise, the input data is sent to NMACS, which uses it to find and a component capable of carrying out the functionality. In either case, the selected component is used by the middleware, which uses the data returned by the component to update the system state.

It should be assumed that the middleware:

- 'Remembers' (stores a reference to the location of) the component designed for this task
- Has an appropriate range of test data to send to NMACS, with which NMACS can test a candidate component for compatibility/suitability
- Is always supplied with test data that are sufficiently precise and detailed to find an appropriate near match. Test data may consist of concrete values (i.e. constants) or ranges (inclusive sequences of constants)

7.5.2 Page States and User Data Validation

Although the state of a page as a whole can initially be decided from default values when first constructed, the page state (and thus the page's appearance) may change as the user makes choices. As has been discussed, the local (GUI wizard) system reads and stores user input, the decision-making process that determines state is done remotely, and the middleware provides access to the decision-making process for the wizard.

When the 'Ok' button is pressed, the local system prompts the middleware, which then attempts to use the component pre-defined at design-time to obtain a validation message for this page. If this component is no longer available, then (NM)ACS is used in order to find a component that returns a validation message for this page.

If the validation message is empty (i.e. an empty string), then it is assumed that the page is valid. Otherwise, it is treated as an error message. In either case, the message is stored so that it can be used by the wizard to determine the validation status and (if applicable) the error message to display. The user will only be allowed to progress to the next page if the validation status is OK, i.e. if all the choices the user has made are valid.

7.5.3 Storage of System State

It is assumed that each widget is a variable in itself – in other words, that each widget stores the data entered into it (text fields) or selection made on it (combo-boxes, radio buttons and checkboxes). User selection data is supplied by the wizard to the middleware, which uses it to maintain the system's state. The system state consists of variables that reflect user data, and also of inferred variables such as whether an option should be active or inactive. The middleware typically uses components to get the values of inferred variables.

The following table shows each of the "user data" state variables, together with its data type, a brief description of its purpose and the page on which it appears.

State variable	Type	Description	Page
IS_USER_REGISTERED	Boolean	Whether user is registered	1
USER_NAME	String of text	The user's user-name	1
USER_PASSWORD	String of text	The user's password	1
DEPART_STATION	String of text	The departure station	2
DEST_STATION	String of text	The destination station	2
IS_RETURN_TICKET	Boolean	Whether the user wants a return ticket	2
IS_TICKET_ECONOMY	Boolean	Economy class ticket type requested	3
IS_TICKET_STANDARD	Boolean	Standard class ticket type requested	3
IS_TICKET_BUSINESS	Boolean	Business class ticket type requested	3

The next table shows the same details for each of the inferred state variables.

Variable	Type	Description	Page
IS_USER_NAME_ACTIVE	Boolean	Whether the "username" field should be active	1
IS_PASSWORD_ACTIVE	Boolean	Whether the "password" field should be active	1
TICKET_PRICE	Decimal	The price of the requested ticket, in pounds sterling	4
ERROR_MESSAGE	String	The result of the previous validation	1, 2, 3

The validation variable "MESSAGE" is a special state variable. It is only updated when a validation request for a page is sent by the wizard to the middleware. The middleware then updates the validation variable as appropriate.

7.5.4 Deciding Page States

As has been said, the state of a page is decided by reference to a remote component, which is accessed via the (NM)ACS process. Assuming that a page is valid, the rules determining the next page's construction when 'Ok' is pressed is as follows:

1. User clicks 'Ok' and the page is valid
2. Wizard asks middleware to update system state
3. Middleware uses components to update system state
4. Next page is drawn by the wizard depending on the updated state

In addition to being bound to a "user selection" state variable, a widget may have an own activation state which is bound to an "inferred" state variable. The activation state variable can be true (active) or false (inactive).

Inferred state variables are affected by the values of one or more state variables. For example, the "is user registered" checkbox on page 1 determines whether the "user-name" and "password" fields are active or inactive. The widget can thus in effect be made active or inactive on-screen (by the wizard) whenever another widget's data or selection is changed (by the middleware).

7.5.5 Middleware Functions

The middleware contains functions to update the system state based on new user selection data, and functions to validate page states and to provide validation information. They are listed in the table below.

Variable	Input(s)	Output(s)	Description
updateState	Attribute, Boolean	(None)	Updates the state with the new Boolean value for the given attribute.
updateState	Attribute, Text string	(None)	Updates the state with the new text value for the given attribute.
updateState	Attribute, Decimal	(None)	Updates the state with the new decimal value for the given attribute.
updateState	Attribute, Integer	(None)	Updates the state with the new integer value for the given attribute.
isPageValid	Page number	Is valid	Validates the given page number and returns whether that page is valid.
getErrorMsg	(None)	Validation error message	Returns “None” if the last page validation was successful, or a message detailing the error that occurred if not.

Attributes are tokens representing each state variable. The names and details of these attributes were given in section 7.5.3.

7.5.6 The Components in the Repository

It is assumed that the middleware knows:

- When access to external functionality is required
- The data necessary for the (NM)ACS process
- How to invoke the (NM)ACS process
- The names of all the components available in the repository

- The names of all the components the system was designed to use
- How to use the selected component
- How to interpret and use the results returned by the selected component

This section outlines the remote components available to the middleware. First, the components that the system is designed to use if available (i.e. before the (NM)ACS process is needed) are detailed. These are the components that the system was intended to use as standard. Second, the components that (NM)ACS has access to are detailed. These are the alternative components available in the repository, which may be capable of the necessary functionality.

Appendix A gives the details of each component that the system is designed to use. Each component corresponds to the table of functionality detailed in the previous subsection. For each component listed in Appendix A, the following details are given:

- The technical name (i.e. the ID)
- A brief, human-readable description
- The input types and the internal names used for these types, in the order in which each type is expected. These are shown in the format <*internal-name*> as <*type*>
- The return (output) type
- The business logic, in a pseudocode format
- Test data. This includes a number of {input → output} pairs, where input and output parameters are single values (e.g. false, 34, 9.9)

Note that, in some circumstances, it is impossible to reproduce appropriate test data. This is because test data must test for all of the boundary values in all available data ranges, in addition to all cases in which data are handled in special ways (i.e. exceptions). If it is impossible to produce an exhaustive list of possible outcomes, then the data cannot be produced. Failure to include all such details would impair the reliability of the component if it was selected for use, given its importance in the (NM)ACS process.

7.5.7 The Structure of the Repository

For the sake of simplicity, clarity and brevity, each component contains only one externally visible function[33] (because it is expected to be capable of one task). Components are wrapped within objects called *component containers*. A container contains the following details:

- The name of the component (i.e. the component's ID)
- The java class that contains the component's functionality (i.e. the technical name associated with the ID)
- Test data associated with the component (which is used to test the component's externally visible function)

[33] An externally visible function is equivalent to a Java method whose signature contains the `public` modifier.

A component container can contain multiple sets of test data. Each individual test data set consists of:

- A collection of input data types and values and/or value ranges
- The output type and the expected value

Input ranges are defined in one of the range classes provided with the source code for this example.

7.5.8 (NM)ACS in Use

The remainder of this case study will only concentrate on situations where the middleware sends input data to (NM)ACS, which finds an appropriate component (see shaded area in the Figure 34). This will show how components can be selected and substituted. This process can be examined in order to show how effective (NM)ACS is at meeting aims 1, 2, 3, 4, 6, 7 and 8, as explained in section 1.7. The analysis of the case study in achieving these aims is given in section 7.7.

As for the details necessary for the (NM)ACS process, the input data will contain:

- The types and values of the data to send into the remote component as inputs
- The data type expected back from the component
- Test input and output data

To reiterate, the stages involved in the match process are as follows. This section as a whole covers aims 1, 4, 6, 7 and 8. The additional individual aims that each stage intends to evaluate the case study against are given in brackets.

1. Attempt to find an exact match (aim 2) (see Figure 15 and Figure 16). If ID, version, functions and I/O match, then exact match found - use component. (Note that ‘dynamic’ exact matching (chapter

4.2.2.2) is not used here, as it is assumed that there are no prior preferences additional to those deemed necessary for the (NM)ACS process)

2. Otherwise, attempt to find a near match (aim 3) (see Figure 19 and Figure 20):

    2.1. Produce list of component functions whose input and output types match those needed

    2.2. Restrict this list so that it only includes functions that match the test data/exceptions supplied by the previously used component

In finding an exact match, (NM)ACS matches the ID of the component and the signature of each component function (i.e. signature matching). For a near match, the input and output types, their ordering in the signature and the test using test data must match. Testing of the case study must thus demonstrate how (NM)ACS might work in tests where:

- An exact match is available (aim 2) (see section 7.5.8.1)
- A near match is available (aim 3):
    - A candidate component uses identical signature types, and returns appropriate and inappropriate results (see section 7.5.8.2)
    - A candidate component uses different signature types, and returns appropriate and inappropriate results (see section 7.5.8.3)

The next subsection covers the possible scenarios where replacement components would be needed and how exact- and near-match selection would operate, for each of the above bullet points. Technical details concerning the source code are given in Appendix A and Appendix B.

7.5.8.1 Exact Match

Scenario: The instance of component isUserNameRequired used by the middleware is no longer available.

Test: A duplicate component (Component 13) is available.

Stage 1: Produce list of exact match components, using ID 'isUserNameRequired'

Result: Exact match component found.

7.5.8.2 Component with Identical Signature Types and Order

Scenario: The instance of component isPasswordRequired used by the middleware is no longer available. (Assume that neither instance of component isUserNameRequired is available.)

Test: isBusinessClassAvailable (Component 7) is functionally capable of being an alternative component. Although isBusinessClassOptionHidden (Component 12) contains an identical signature and type order, the returned value is incorrect.

Stage 1: Produce list of exact match components, using the component ID 'isPasswordRequired'

No components in list → use near-match (Stage 2)

Stage 2.1: Produce list of components where the input type list matches the list {boolean} and the output type list matches the list {boolean}

Components in list: isBusinessClassAvailable, isBusinessClassOptionHidden

Stage 2.2: From this list, find first component matching test data/exceptions

isBusinessClassAvailable: test results: {true → true}, {false → false} – match

isBusinessClassOptionHidden: test results: {true → false}, {false → true} – no match

Result: Near-match component found: isBusinessClassAvailable

7.5.8.3 Component with Different Signature Types

Scenario: The instance of component validateUser used by the middleware is no longer available.

Test: validateUserDetails (Component 14) uses a different signature, but contains the appropriate functionality.

Stage 1: Produce list of exact match components, using the component ID 'validateUser'

No components in list → use near-match (Stage 2)

Stage 2.1: Produce list of components where the input type list matches the list {boolean, String, String} and the output type list matches the list {String}

Components in list: [None]

Stage 2.2: From this list, find first component matching test data/exceptions

Component found: [None]

Stage 2.3: Produce a list of components where the input type list matches the list {boolean, {String, character array}, {String, character array}} and the output type list matches the list {{String, character array}}

Components found: [None]

Result: No components found.

7.6 Interpreting the Results - Overview

In none of the tests was an inappropriate component selected by (NM)ACS. This is a promising sign that shows the ability of (NM)ACS to select components, assuming that the case study used is fair and comprehensive enough to give a precise account.

However, the tests also show that (NM)ACS is insufficient in its presently described state, in some scenarios, at selecting a viable component. Examples of these are where components with relevant functionality either use the same signature types as the test data but in a different order. This could be solved by losing the rigidity currently present in identical type ordering. However, this would be difficult, given the complexities involved in replacing one type with another beyond the strategies already being used.

7.7 Evaluation of the (NM)ACS Approach

As discussed, the aims of this work (in addition to evaluating Web technologies, which is covered in chapter 6) are to show that both exact and near component match is viable. Goal Question Metrics (GQM) is the approach to be used to evaluate the case study against these criteria.

GQM is a requirements evaluation method that allows goals to be measured. It offers a way of systematically defining, establishing and exploiting measurement methods in order to evaluate software processes and products. It also provides the ability to create measurable metrics, which can be used in the evaluation.

Firstly, *goals* have to be defined, which were covered in section 1.7. Secondly, *questions* need to be expressed that, when answered, show whether the goals have been achieved. These will be expressed

in section 7.7.1, as an adaptation of these goals. Thirdly, *metrics* are applied in order to answer the questions. This is performed in section 7.7.2 (Benedicenti et al, 1996).

In GQM, a measurement process is necessary in order to articulate the goals. This will be the case study itself. The goals need to be represented as data – a measurement program – which will be the results of the case study. These data are of two kinds. The first is the price that is calculated on the last screen of the Wizard. This will verify that components were used appropriately and the method was accurate. The second is the means of calculating the price – that is, on-screen output that shows the way in which (NM)ACS was carried out. Finally, in GQM, a way of interpreting the data needs to exist in order to determine whether the goals have been achieved. The price that the Wizard calculates can readily be verified using the table of ticket prices. Similarly, a human observer can verify the on-screen output (Pressman, 1997; Fuggetta et al, 1998).

7.7.1 Questions

The following questions are constructed in order to determine whether the aims explained in section 1.7 have been achieved. Therefore, each question refers to each aim in turn.

Question 1: Can candidate components be selected automatically?

Question 2: Can exact-match components ably be selected?

Question 3: Can near-match components ably be selected?

Question 4: Can components ably be substituted?

Question 5: Does one or more Web technologies exist to facilitate (NM)ACS?

Question 6: Is there a selection strategy to exemplify how (NM)ACS could be carried out?

Question 7: Can (NM)ACS be used with existing software technologies?

Question 8: Can (NM)ACS be shown to be successful using a number of case studies?

7.7.2 Metrics

This section will show how well the (NM)ACS process, the suggested (NM)ACS strategy and the case studies achieve the aims. An answer of 'yes' means that an aim was achieved, and an answer of 'no' means that it was not.

Question 1: Yes. Candidate components are selected automatically (see sections 7.5.8.1 and 7.5.8.2).

Question 2: Yes. The case study demonstrates that, where an exact-match candidate component is available, it is selected automatically (see section 7.5.8.1).

Question 3: Yes. The case study demonstrates that, where an exact-match candidate component is not available, a near-match component can be selected automatically where available (see section 7.5.8.2).

Question 4: Yes. In the case study, components are substituted and the new component is then used in order to determine the ticket price (see sections 7.5.8.1 and 7.5.8.2).

Question 5: Yes. According to the evaluation of Web technologies (see chapter 6), EJB, WSDL and DCOM are all capable of facilitating (NM)ACS.

Question 6: Yes. See the selection strategy explained in chapter 4.2.2.

Question 7: Yes. The evaluation of Web technologies shows that (NM)ACS can be used within a number of frameworks. Additionally, the case study shows that (NM)ACS can be implemented using currently available hardware and software technologies.

Question 8: Yes. See earlier on in this chapter.

The metrics above show that all of the aims were achieved. This observation holds to the extent of the evaluation of Web technologies, the suggested selection strategy and the case study.

7.8 Limitations

Upon carrying out the case study, a number of issues became evident additional to the limitations explained in section 1.5. Possible solutions will be discussed in section 7.9.

1. If a component's business logic changes, then the test data needs to be updated and the system using the component needs to update this data

2. Sometimes, test data are impossible to produce. This may reduce the success with which NMACS finds components that it considers appropriate

3. Greater component complexity means less chance of finding a match

4. If test data are too detailed, may as well use test data to obtain results and not the component! (See GetStations)

5. Arrays are sometimes difficult to put into test data sets. For example, in component GetStations, there are eight valid combinations of returned station names, and that's assuming there are only ever three station names to return

6. Strings are sometimes difficult to put into test data sets. For example:

    - Case sensitivity may not matter (e.g. "Stoke" may be as valid as "STOKE")

- A particular message may be spelled differently but be equally valid in two different components doing the same job (e.g. “Invalid destination station” may be as valid a message as “Destination station is invalid”)

7. Sometimes it is impossible to provide test data set because valid inputs cannot be predicted. This may occur, for example, when the result depends on the contents of a database that continually change

8. There is no facility for near-matching components where the signatures match but the inputs are taken in a different order

7.9 Improvements for the Future

A solution for limitation 1 might be to retrieve the test data from a source that recalculates values as and when necessary, depending on data (such as in a database) that may change with time (such as prices), rather than storing a hard-coded value.

An improvement for limitation 3 might be to relax the signature match rules for near-matching. This could be done by ignoring the ordering of the signature types. If this was done, Component 13 might have been deemed a suitable replacement component.

A way of specifying the relevancy of the order of array elements in test data sets might prove useful. For example, a collection of data might be processed (e.g. iterated over) in a way that makes their ordering in the signature irrelevant.

A way of specifying the relevancy of case sensitivity of strings in test data sets might prove useful in improving the scope of matches, as well as speeding up the process if case insensitive.

Where it is impossible to provide test data set because valid inputs cannot be predicted, it may be possible (at performance cost) for another component or database to be dynamically called at the time where the test data are needed. In the case where another component is used, circularity problems would need to be guarded against.

In cases where no alternate components can be found, a default return value can be specified with the test data. For example, if no replacement component can be found for IsEconomyClassAvailable, then the desired default return value may be false (i.e. economy class should not be available by default).

A way of checking signature compatibility regardless of the order of inputs might also improve the scope of matches.

Finally, a candidate component may be compatible if its signature uses 'stronger' (i.e. more precise) types than those in the signature of the component being replaced. For example, if the previous component's signature accepted a (32-bit) integer variable, then a component with a signature containing a (64-bit) long variable should be acceptable. In the case of values being returned, the value can be reduced in precision, such as by truncation or nearest value, to fit the expected type.

8 Discussion

It was mentioned in the previous chapter that all scoring carried out was subjective, drawing from an understanding of, and thus being limited to the depth of detail of, the documentation that has been considered. In light of this, the results can only suggest an answer by giving an educated opinion about the quality, performance and ability of each technology given the documentation available. By increasing the number of candidates and testing each one instead of relying on the documentation available, the evaluation would perhaps give a better picture of suitability of each candidate. However, this would be at the cost of substantial additional time and effort.

Each of the Web technologies obtained similar results in the feature analysis. Possible reasons for this are that they are very similarly able to facilitate (NM)ACS, or that the feature analysis was insufficiently detailed to show a clear comparison. Both questions can be addressed only by improving upon the analysis in the ways that are mentioned above.

This document observes the minimum requirements that are possible for a generic distributed component-based system. However, there are a number of factors that have not been accounted for. These include, for example, fault tolerance (the ability of the system to cope with unexpected faults), security (the provision of security measures such as encryption) and issues such component (functionality) ownership and licensing. Clearly, these areas are of considerable importance to particular users of distributed systems, such as intelligence agencies and safety-critical organisations.

It would be possible to progress the ACS framework and description with one or more of these factors in mind. However, this may mean that it will have to change dramatically, and as a consequence become much more complicated. This research has also been dependent upon a number of assumptions such as the availability of an appropriate number of components and the ability for functionality to be used across different platforms, which limit the usefulness of the results that have been made.

9 Conclusion

It has been shown that the aims as explained in section have been achieved, at least to the extent that the evaluation of Web technologies, the suggested selection strategy and the case study. Exact- and near-match ACS are both possible, given the correct component descriptions and reuse strategy. A set of properties has been suggested, which represents the minimum requirements necessary for an NMACS description. The suggested component selection policies were inspired by four possible scenarios in which components would need to be replaced. These policies included searching for both “exact” and “near” component matches.

In the event that an exact match cannot be found, a near-match policy is adopted. This policy uses the descriptions suggested and matches property values of the component to be substituted with those of potential replacement components, via the static behaviour handling matching method. If no appropriate components are identified at this stage, components are tested in their ability to provide some given functionality for suitability. This is carried out by operating the appropriate function with the input property value(s), and ascertaining whether the function's output matches the corresponding output property value(s).

All of the properties described in the NMACS description can be gathered automatically, and used automatically in a component matching strategy such as the substitution policies that have been identified. The ability to automate completely the process of substituting components was thus illustrated, as was the ability to obtain components that are able to carry out particular functionality regardless of whether they were advertised for that explicit purpose.

Web technologies exist to allow description languages to be implemented. A number of Web technologies were evaluated in their ability to facilitate the NMACS description, within the framework of the proposed substitution method. The availability of at least one appropriate candidate shows that

complete automation of the component description and searching process is possible given the Web technologies that currently exist.

The results show that WSDL is slightly more probable a candidate than either EJB or DCOM for (NM)ACS. Its flexible Web services infrastructure ensured its success over the other two due to the degree with which its flexibility enabled (NM)ACS to be implemented.

This work suggests a description and a use strategy for (NM)ACS and identifies a number of Web technologies that can facilitate this strategy. Therefore, the "nirvana of complete automation with no loss of descriptive precision" described in section 3.1 appears to have been reached. However, a number of assumptions and limitations were made in order to ensure clarity and to avoid the focus becoming too large. The suggested description and use strategy might be improved by expanding on the research, for example to include literature in the fields of fault tolerance, data security and component licensing. Addressing the limitations of signature parameter ordering and lack of support for parameters that are objects or ranges, as discussed in section 1.5, as well as candidate components with signature types more precise (and thus castable[34]), would also improve (NM)ACS. These improvements might in turn affect the overall requirements for (NM)ACS and thus the features necessary for the evaluation. Furthermore, the number of candidate Web technologies can be increased, which will improve the scope with which appropriate Web technologies can be identified through the feature analysis.

10 Appendices

Appendix A List of Components in the Repository

Component 1: IsUserNameRequired

Description: "Returns whether a user-name is required."
Input type(s): isUserRegistered as boolean
Return type: boolean
Pseudocode:
if (isUserRegistered = true) *then* return **true**
else return **false**

Test data: {true → true}, {false → false}

Component 2: IsPasswordRequired

Description: "Returns whether a password is required."
Input type(s): isRegistered as boolean
Return type: boolean
Pseudocode:
if (isRegistered = true) *then* return **true**
else return **false**

Test data: {true → true}, {false → false}

Component 3: ValidateUser

Description: "Validates the user's details"
Input type: isRegistered as boolean, username as String, password as String
Return type: String
Pseudocode:
if (isRegistered = false) *then* return **"" (empty String)**
else if (username/password combination is valid) *then* return **"" (empty String)**
else return **"Invalid user-name/password combination"**

Test data: N/A – it is impossible to predict valid user-name/password combinations

[34] This word is used here in the object-oriented context to mean any type to which a smaller type can be converted, or *cast*, without losing data (i.e. precision).

Component 4: GetStations

Description: "Returns the list of available train stations"
Input type: <none>
Return type: String array
Pseudocode:
return {**"Stoke"**, **"Stafford"**, **"Blythe Bridge"**}

Test data: {"Stoke", "Stafford", "Blythe Bridge"}

Component 5: ValidateDepartureAndDestination

Description: "Validates the departure station and destination station"
Input type: departureStation as String, desinationStation as String
Return type: String
Pseudocode:
if (departureStation is not valid) *then* return **"Invalid departure station"**
else if (destinationStation is not valid) *then* return **"Invalid destination station"**
else if(departureStation = destinationStation) then:
return **"Departure and destination stations must be different"**
else return **"" (empty String)**

Test data: {"Stoke", "Stafford" → ""}, {"Incorrect", "Stafford" → "Invalid departure station"}, {"Blythe Bridge", "Incorrect" → "Invalid destination station"}, {"Stoke", "Stoke" → "Departure and destination stations must be different"}

Component 6: IsEconomyClassAvailable

Description: "Returns whether economy class tickets are available to a given user"
Input type: isRegistered as boolean, userName as String
Return type: boolean
Pseudocode:
if (isRegistered = true, and userName is a valid user, and user is a student or OAP)
then return **true**
else return **false**

Test data: N/A – impossible to predict list of valid user-names

Component 7: IsBusinessClassAvailable

Description: "Returns whether business class tickets are available to a given user"
Input type: isRegistered as boolean
Return type: boolean
Pseudocode:
if (isRegistered = true) *then* return **true**
else return **false**

Test data: {true → true}, {false → false}

Component 8: IsTicketSelected
Description: "Returns whether one of the three ticket types has been selected."
Input type: isEconomy as boolean, isStandard as boolean, isBusiness as boolean
Return type: boolean
Pseudocode:
if (isEconomy = true or isStandard = true or isBusiness = true) *then* return **true**
else return **false**
Test data: {true, false, false → true}, { false, true, false → true}, {false, false, true → true}, {false, false, false → false}

Component 9: GetTicketPrice
Description: "Calculates and returns the ticket price, based on given details."
Input type: departStation as String, destStation as String, isReturn as boolean, isEconomy as boolean, isBusiness as boolean,
Return type: decimal (two decimal places)
Pseudocode:
result = price of a single ticket from departStation to destStation, in pounds sterling
if (isReturn = true) *then* result = result * 1.75
if (isEconomy = true) *then* result = result * 0.5
if (isBusiness = true) *then* result = result * 2.0
return result
Test data: {"Stoke", "Stafford", false, false, false → 8.00}, {"Stafford", "Blythe Bridge", true, false, false → 17.50}, {"Blythe Bridge", "Stafford", true, true, false → 8.75}, {"Stoke", "Blythe Bridge", false, false, true → 11.00}

The next set of tables gives similar details for the other components available in the repository.

Component 10: GetTicketPriceInDollars
Description: "Determines ticket price, in dollars."
Input type: isReturn as boolean, departStation as String, destStation as String, isEconomy as boolean, isBusiness as boolean
Return type: decimal (two decimal places)
Pseudocode:
result = price of a single ticket from departStation to destStation, in dollars
if (isReturn = true) *then* result = result * 1.75
if (isEconomy = true) *then* result = result * 0.5
if (isBusiness = true) *then* result = result * 2.0
return result
Test data: {false, "Stoke", "Stafford", false, false → 14.55}, {true, "Stafford", "Blythe Bridge", false, false → 31.82}, {true, "Blythe Bridge", "Stafford", true, false → 15.91}, {false, "Stoke", "Blythe Bridge", false, true → 20.00}

Component 11: GetTicketPriceInPounds

Description: "Determines ticket price, in pounds sterling."
Input type: isReturn as boolean departStation as String, destStation as String, isEconomy as boolean, isBusiness as boolean
Return type: decimal (two decimal places)
Pseudocode:

result = price of a single ticket from departStation to destStation, in pounds sterling
if (isReturn = true) *then* result = result * 1.75
if (isEconomy = true) *then* result = result * 0.5
if (isBusiness = true) *then* result = result * 2.0
return result

Test data: {false, "Stoke", "Stafford", false, false → 8.00}, {true, "Stafford", "Blythe Bridge", false, false → 17.50}, {true, "Blythe Bridge", "Stafford", true, false → 8.75}, {false, "Stoke", "Blythe Bridge", false, true → 11.00}

Component 12: IsBusinessClassOptionHidden

Description: "Returns whether business class tickets are hidden from the user"
Input type: isRegistered as boolean
Return type: boolean
Pseudocode:

if (isRegistered = true) *then* return **false**
else return **true**

Test data: {true → false}, {false → true}

Component 13: IsUserNameRequired2 [duplicate of Component 1, above]

Description: "Returns whether a user-name is required."
Input type(s): isRegistered as boolean
Return type: boolean
Pseudocode:

if (isRegistered = true) *then* return **true**
else return **false**

Test data: {true → true}, {false → false}

Component 14: ValidateUserDetails

Description: "Validates user information. Returns a message describing the result."
Input type: username as String, password as String
Return type: String
Pseudocode:

if (username or password not supplied) *then* return **"" (empty String)**
else if (username/password combination is valid) *then* return **"" (empty String)**
else return **"Invalid user-name/password combination"**

Test data: N/A – it is impossible to predict valid user-name/password combinations

Component 15: GetPriceOfTicket

Description: "Determines and returns ticket price."
Input type: departStation as character array, destStation as character array, isReturn as boolean, isEconomy as boolean, isBusiness as boolean
Return type: double *(n.b. maximum value is twice as big as standard decimal)*
Pseudocode:

result = price of a single ticket from departStation to destStation, in dollars
if (isReturn = true) *then* result = result * 1.75
if (isEconomy = true) *then* result = result * 0.5
if (isBusiness = true) *then* result = result * 2.0
return result

Test data: {false, "Stoke", "Stafford", false, false → 8.0}, {true, "Stafford", "Blythe Bridge", false, false → 17.5}, { true, "Blythe Bridge", "Stafford", true, false → 8.75}, {false, "Stoke", "Blythe Bridge", false, true → 11.0}

Appendix B Source Code Listing of Components in the Repository

Each component may contain a number of methods, but only contains one public method. (NM)ACS finds this public method via reflection, and uses it as the entry point into the component. Components only return wrapper objects (rather than primitive types). Typically, components in this repository follow the convention that the public method has the same name as the class, except that the first character is lower-case. However, this is not necessary, and some components do not follow the convention (in order to try and catch NMACS out and test the programmatic integrity of the process). Each component listed here corresponds to a component listed in Appendix A.

Component 1: IsUserNameRequired

```
package repository.components;

/**
 * IsUserNameRequired v1.0<BR>
 * Created 17-Jul-2005, 18:06:05<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsUserNameRequired {
  /**
   * Returns whether a user-name is required.
   *
   * @param isUserRegistered  whether the user is registered.
   * @return  whether a user-name is required.
   */
  public Boolean execute(Boolean isUserRegistered) {
    if (isUserRegistered.booleanValue() == true)
      return new Boolean(true);
    else
      return new Boolean(false);
  }
}
```

Component 2: IsPasswordRequired

```
package repository.components;

/**
 * IsPasswordRequired v1.0<BR>
 * Created 17-Jul-2005, 18:06:05<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
```

```
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsPasswordRequired {
  /**
   * Returns whether a password is required.
   *
   * @param isUserRegistered  whether the user is registered.
   * @return  whether a password is required.
   */
  public Boolean isPassRequired(Boolean isUserRegistered) {
    if (isUserRegistered.booleanValue() == true)
      return new Boolean(true);
    else
      return new Boolean(false);
  }
}
```

Component 3: ValidateUser

```
package repository.components;

/**
 * ValidateUser v1.0<BR>
 * Created 14-Aug-2005, 18:28:10<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class ValidateUser {
  /**
   * Validates the user's details.
   *
   * @param isRegistered  whether the user has specified that he/she is registered.
   * @param username      the user-name that the user entered.
   * @param password      the password that the user entered to correspond with the
   *                      given user-name.
   *
   * @return  a message, which is the result of validating the given details.
   */
  public String validateUser(boolean isRegistered, String username, String password) {
    if(!isRegistered)
      return "";
    else if(isValid(username, password))
      return "";
    else
      return "Invalid user-name/password combination";
  }
  /**
   * Returns whether the given username and password combination is valid, which
   * for the purposes of this example is always <code>true</code>.
   *
   * @param username  the user-name that the user entered.
   * @param password  the password that the user entered to correspond with the
   *                  given user-name.
   *
   * @return  whether the given username and password combination is valid, which
   *          for the purposes of this example is always <code>true</code>.
   */
  private boolean isValid(String username, String password) {
    return true;
  }
}
```

Component 4: GetStations

```
package repository.components;

import util.Constants;

/**
 * GetStations v1.0<BR>
 * Created 14-Aug-2005, 18:34:59<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class GetStations {
  /**
   * Returns the list of available train stations.
   *
   * @return  Returns the list of available train stations
   */
  public String[] getStations() {
    return Constants.STATION_NAMES;
  }
}
```

Component 5: ValidateDepartureAndDestination

```
package repository.components;

import util.Constants;

/**
 * ValidateDepartureAndDestination v1.0<BR>
 * Created 14-Aug-2005, 18:36:31<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class ValidateDepartureAndDestination {
  /**
   * Validates the departure station and destination station.
   *
   * @param departureStation     the departure station that the user has selected.
   * @param destinationStation   the destination station that the user has selected.
   *
   * @return  whether the given departure and destination stations are valid.
   */
  public String validateDepartureAndDestination(String departureStation,
                                                String destinationStation) {
    if (!isValidDepartureStation(departureStation))
      return "Invalid departure station";
    else if(!isValidDestinationStation(destinationStation))
      return "Invalid destination station";
    else
      return "";
  }

  /**
   * Returns whether the given departure station is contained within the list of
   * station names given in the <code>Constants</code> class.
   *
   * @param departureStation  the departure station to check for validity.
```

```
   * @return  whether the given departure station is contained within the list of
   *          station names given in the <code>Constants</code> class.
   *
   * @see Constants#STATION_NAMES
   */
  private boolean isValidDepartureStation(String departureStation) {
    for(int i=0; i<Constants.STATION_NAMES.length; i++) {
      if(Constants.STATION_NAMES[i].equalsIgnoreCase(departureStation)) {
        return true;
      }
    }

    return false;
  }

  /**
   * Returns whether the given destination station is contained within the list of
   * station names given in the <code>Constants</code> class.
   *
   * @param destinationStation  the destination station to check for validity.
   * @return  whether the given destination station is contained within the list of
   *          station names given in the <code>Constants</code> class.
   *
   * @see Constants#STATION_NAMES
   */
  private boolean isValidDestinationStation(String destinationStation) {
    for(int i=0; i<Constants.STATION_NAMES.length; i++) {
      if(Constants.STATION_NAMES[i].equalsIgnoreCase(destinationStation)) {
        return true;
      }
    }

    return false;
  }
}
```

Component 6: IsEconomyClassAvailable

```
package repository.components;

/**
 * IsEconomyClassAvailable v1.0<BR>
 * Created 14-Aug-2005, 18:42:18<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsEconomyClassAvailable {
  /**
   * Returns whether economy class tickets are available to a given user. Economy
   * class tickets are available to registered users who are students and/or OAPs.
   *
   * @param isRegistered  whether the user specifies that he/she is registered.
   * @param userName      the user-name that the user supplied.
   *
   * @return  Returns whether economy class tickets are available to a given user.
   */
  public Boolean isEconomyClassAvailable(Boolean isRegistered, String userName) {
    if(isRegistered.booleanValue() && isValidUserName(userName) && isConcession(userName))
      return new Boolean(true);
    else
      return new Boolean(false);
  }
```

```
  /**
   * Returns whether the given user-name is valid, which for the purposes of this
   * example is always <code>true</code>.
   *
   * @param userName  the user's user-name.
   * @return  whether the given user-name is valid, which for the purposes of this
   *          example is always <code>true</code>.
   */
  private boolean isValidUserName(String userName) {
    return true;
  }

  /**
   * Returns whether the user registered with the given user-name is a student and/or OAP,
   * which for the purposes of this example is always <code>true</code>.
   *
   * @param userName  the user's user-name.
   * @return  whether the user registered with the given user-name is a student and/or OAP,
   *          which for the purposes of this example is always <code>true</code>.
   */
  private boolean isConcession(String userName) {
    return true;
  }
}
```

Component 7: IsBusinessClassAvailable

```
package repository.components;

/**
 * IsBusinessClassAvailable v1.0<BR>
 * Created 14-Aug-2005, 18:48:44<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsBusinessClassAvailable {
  /**
   * Returns whether business class tickets are available to a given user. Business
   * class tickets are available to registered users.
   *
   * @param isRegistered  whether the user specifies that he/she is registered.
   * @param userName      the user-name that the user supplied.
   *
   * @return  Returns whether business class tickets are available to a given user.
   */
  public Boolean isBusinessClassAvailable(Boolean isRegistered, String userName) {
    if(isRegistered.booleanValue() && isValidUserName(userName))
      return new Boolean(true);
    else
      return new Boolean(false);
  }
  /**
   * Returns whether the given user-name is valid, which for the purposes of this
   * example is always <code>true</code>.
   *
   * @param userName  the user's user-name.
   * @return  whether the given user-name is valid, which for the purposes of this
   *          example is always <code>true</code>.
   */
  private boolean isValidUserName(String userName) {
    return true;
  }
}
```

Component 8: IsTicketSelected

```
package repository.components;

/**
 * IsTicketSelected v1.0<BR>
 * Created 14-Aug-2005, 20:15:11<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsTicketSelected {
  /**
   * Returns whether one of the three ticket types has been selected.
   *
   * @param isEconomy   whether an economy class ticket has been selected by the user.
   * @param isStandard  whether a standard class ticket has been selected by the user.
   * @param isBusiness  whether a business class ticket has been selected by the user.
   *
   * @return  whether one of the three ticket types has been selected.
   */
  public Boolean isTicketSelected(Boolean isEconomy,
                                  Boolean isStandard,
                                  Boolean isBusiness) {
    return isEconomy.booleanValue()
        || isStandard.booleanValue()
        || isBusiness.booleanValue();
  }
}
```

Component 9: GetTicketPrice

```
package repository.components;

import util.Constants;

/**
 * GetTicketPrice v1.0<BR>
 * Created 14-Aug-2005, 20:19:12<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class GetTicketPrice {
  /**
   * Calculates and returns the ticket price, based on given details. The value
   * returned is to two decimal places.
   *
   * @param departStation  the station from which the user wants to depart.
   * @param destStation    the station to which the user wants to go.
   * @param isReturn       whether the ticket is a return ticket.
   * @param isEconomy      whether an economy class ticket was requested.
   * @param isBusiness     whether a business class ticket was requested.
   *
   * @return  the ticket price, based on given details.
   */
  public Double getTicketPrice(String departStation, String destStation,
                               Boolean isReturn, Boolean isEconomy,
                               Boolean isBusiness) {
    Double result = getStandardPrice(departStation, destStation);
```

```
    if(isReturn.booleanValue())
      result = result * 1.75;

    if(isEconomy.booleanValue())
      result = result * 0.5;
    else if(isBusiness.booleanValue())
      result = result * 2.0;

    return toTwoDecimalPlaces(result);
  }

  /**
   * Returns the price of a standard ticket from the gievn departure station to the given
   * destination station, in pounds sterling.
   *
   * @param departStation  the station from which the user wants to depart.
   * @param destStation    the station to which the user wants to go.
   * @return  the price of a standard ticket from the gievn departure station to the given
   *          destination station, in pounds sterling.
   */
  private Double getStandardPrice(String departStation, String destStation) {
    // Stoke to...
    if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(departStation)) {
      // ...Stafford
      if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(destStation))
        return Constants.PRICE_STOKE_TO_STAFFORD;
      // ...Blythe Bridge
      else
        return Constants.PRICE_STOKE_TO_BLYTHE;
    }
    // Stafford to...
    else if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(departStation))
{
      // ...Stoke
      if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
        return Constants.PRICE_STAFFORD_TO_STOKE;
      // ...Blythe Bridge
      else
        return Constants.PRICE_STAFFORD_TO_BLYTHE;
    }
    // Blythe Bridge to...
    else {
      // ...Stoke
      if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
        return Constants.PRICE_BLYTHE_TO_STOKE;
      // ...Stafford
      else
        return Constants.PRICE_BLYTHE_TO_STAFFORD;
    }
  }

  /**
   * Returns the given value to two decimal places.
   *
   * @param value  the value to return to two decimal places (standard rounding).
   * @return  the given value to two decimal places.
   */
  private Double toTwoDecimalPlaces(Double value) {
    return Math.floor((value.doubleValue() * 100.0d) + 0.5d) / 100.0d;
  }
}
```

Component 10: GetTicketPriceInDollars

```
package repository.components;

import util.Constants;

/**
 * GetTicketPriceInDollars v1.0<BR>
 * Created 14-Aug-2005, 20:31:43<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class GetTicketPriceInDollars {
  /**
   * Determines ticket price, in dollars. The value returned is to two decimal
   * places.
   *
   * @param isReturn       whether the ticket is a return ticket.
   * @param departStation  the station from which the user wants to depart.
   * @param destStation    the station to which the user wants to go.
   * @param isEconomy      whether an economy class ticket was requested.
   * @param isBusiness     whether a business class ticket was requested.
   *
   * @return  the ticket price, in dollars.
   */
  public Double getTicketPriceInDollars(Boolean isReturn,
                                        String departStation, String destStation,
                                        Boolean isEconomy, Boolean isBusiness) {
    Double result = getStandardPrice(departStation, destStation);

    if(isReturn.booleanValue())
      result = result * 1.75;

    if(isEconomy.booleanValue())
      result = result * 0.5;
    else if(isBusiness.booleanValue())
      result = result * 2.0;

    return toTwoDecimalPlaces(result);
  }

    /**
   * Returns the price of a standard ticket from the gievn departure station to the given
   * destination station, in American dollars.
   *
   * @param departStation  the station from which the user wants to depart.
   * @param destStation    the station to which the user wants to go.
   * @return  the price of a standard ticket from the gievn departure station to the given
   *          destination station, in American dollars.
   */
  private Double getStandardPrice(String departStation, String destStation) {
    // Stoke to...
    if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(departStation)) {
      // ...Stafford
      if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(destStation))
        return getPoundsToDollars(Constants.PRICE_STOKE_TO_STAFFORD);
      // ...Blythe Bridge
      else
        return getPoundsToDollars(Constants.PRICE_STOKE_TO_BLYTHE);
    }
    // Stafford to...
```

```
    else if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(departStation))
{
      // ...Stoke
      if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
        return getPoundsToDollars(Constants.PRICE_STAFFORD_TO_STOKE);
      // ...Blythe Bridge
      else
        return getPoundsToDollars(Constants.PRICE_STAFFORD_TO_BLYTHE);
    }
    // Blythe Bridge to...
    else {
      // ...Stoke
      if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
        return getPoundsToDollars(Constants.PRICE_BLYTHE_TO_STOKE);
      // ...Stafford
      else
        return getPoundsToDollars(Constants.PRICE_BLYTHE_TO_STAFFORD);
    }
  }

  /**
   * Returns the equivalent value, in American dollars, of the given value in pounds
   * sterling. It is assumed that $1.00 = £0.55.
   *
   * @param pounds  the value, in pounds sterling, to convert to American dollars.
   * @return  the equivalent value, in American dollars, of the given value in pounds
   *          sterling.
   */
  private Double getPoundsToDollars(Double pounds) {
    return new Double(pounds.doubleValue() * (1.0d / 0.55d));
  }

  /**
   * Returns the given value to two decimal places.
   *
   * @param value  the value to return to two decimal places (standard rounding).
   * @return  the given value to two decimal places.
   */
  private Double toTwoDecimalPlaces(Double value) {
    return Math.floor((value.doubleValue() * 100.0d) + 0.5d) / 100.0d;
  }
}
```

Component 11: GetTicketPriceInPounds

```
package repository.components;

import util.Constants;

/**
 * GetTicketPriceInPounds v1.0<BR>
 * Created 14-Aug-2005, 21:35:34<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class GetTicketPriceInPounds {
  /**
   * Determines ticket price, in pounds. The value returned is to two decimal
   * places.
   *
   * @param isReturn       whether the ticket is a return ticket.
   * @param departStation  the station from which the user wants to depart.
   * @param destStation    the station to which the user wants to go.
```

```
  * @param isEconomy      whether an economy class ticket was requested.
  * @param isBusiness     whether a business class ticket was requested.
  *
  * @return  the ticket price, in pounds.
  */
 public Double getTicketPriceInPounds(Boolean isReturn,
                                      String departStation, String destStation,
                                      Boolean isEconomy, Boolean isBusiness) {
  Double result = getStandardPrice(departStation, destStation);

  if(isReturn.booleanValue())
   result = result * 1.75;

  if(isEconomy.booleanValue())
   result = result * 0.5;
  else if(isBusiness.booleanValue())
   result = result * 2.0;

  return toTwoDecimalPlaces(result);
 }

  /**
  * Returns the price of a standard ticket from the gievn departure station to the given
  * destination station, in pounds sterling.
  *
  * @param departStation  the station from which the user wants to depart.
  * @param destStation    the station to which the user wants to go.
  * @return  the price of a standard ticket from the gievn departure station to the given
  *          destination station, in pounds sterling.
  */
 private Double getStandardPrice(String departStation, String destStation) {
  // Stoke to...
  if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(departStation)) {
   // ...Stafford
   if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(destStation))
    return Constants.PRICE_STOKE_TO_STAFFORD;
   // ...Blythe Bridge
   else
    return Constants.PRICE_STOKE_TO_BLYTHE;
  }
  // Stafford to...
  else if(Constants.STATION_NAMES[Constants.STATION_STAFFORD].equalsIgnoreCase(departStation))
{
   // ...Stoke
   if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
    return Constants.PRICE_STAFFORD_TO_STOKE;
   // ...Blythe Bridge
   else
    return Constants.PRICE_STAFFORD_TO_BLYTHE;
  }
  // Blythe Bridge to...
  else {
   // ...Stoke
   if(Constants.STATION_NAMES[Constants.STATION_STOKE].equalsIgnoreCase(destStation))
    return Constants.PRICE_BLYTHE_TO_STOKE;
   // ...Stafford
   else
    return Constants.PRICE_BLYTHE_TO_STAFFORD;
  }
 }

 /**
  * Returns the given value to two decimal places.
  *
  * @param value  the value to return to two decimal places (standard rounding).
  * @return  the given value to two decimal places.
  */
 private Double toTwoDecimalPlaces(Double value) {
```

```
    return Math.floor((value.doubleValue() * 100.0d) + 0.5d) / 100.0d;
  }
}
```

Component 12: IsBusinessClassOptionHidden

```
package repository.components;

/**
 * IsBusinessClassOptionHidden v1.0<BR>
 * Created 14-Aug-2005, 21:37:39<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsBusinessClassOptionHidden {
   /**
    * Returns whether business class tickets are hidden from the user.
    *
    * @param isRegistered  whether the user is registered.
     *
    * @return  whether business class tickets are hidden from the user.
    */
  public Boolean isBusinessClassOptionHidden(Boolean isRegistered) {
    if (isRegistered.booleanValue() == true)
      return new Boolean(false);
    else
      return new Boolean(true);
  }
}
```

Component 13: IsUserNameRequired2

```
package repository.components;

/**
 * IsUserNameRequired2 v1.0<BR>
 * Created 17-Jul-2005, 18:06:05<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class IsUserNameRequired2 {
  /**
    * Returns whether a user-name is required.
    *
    * @param isUserRegistered  whether the user is registered.
    * @return  whether a user-name is required.
    */
  public Boolean execute(Boolean isUserRegistered) {
    if (isUserRegistered.booleanValue() == true)
      return new Boolean(true);
    else
      return new Boolean(false);
  }
}
```

Component 14: ValidateUserDetails

```
package repository.components;

/**
 * ValidateUserDetails v1.0<BR>
 * Created 14-Aug-2005, 21:40:13<BR>
 * (c) Copyright 2005 Michael Alcock.
 * <P>
 * <b>Note:</b> For example simplification purposes, components should only contain one
 * public method. This is picked up via reflection. Also, only objects are acceptable
 * inputs and outputs, to allow easier storage and manipulation of test data.
 */
public class ValidateUserDetails {
  /**
   * Validates user information. Returns a message describing the result.
   *
   * @param username  the user-name that the user entered.
   * @param password  the password corresponding to the user-name that the user entered.
   *
   * @return  a message describing the result of validating the user information.
   */
  public String validateUserDetails(String username, String password) {
    if(isEmpty(username) || isEmpty(password))
      return "";
    else if(isValidUser(username, password))
      return "";
    else
      return "Invalid user-name/password combination";
  }

  /**
   * Returns whether the given input is null or an empty string.
   *
   * @param input  the input to evaluate.
   * @return  whether the given input is null or an empty string.
   */
  private boolean isEmpty(String input) {
    return input == null || (input = input.trim()).equals("");
  }

  /**
   * Returns whether the given username and password combination is valid, which
   * for the purposes of this example is always <code>true</code>.
   *
   * @param username  the user-name that the user entered.
   * @param password  the password corresponding to the user-name that the user
   *                  entered.
   * @return  whether the given username and password combination is valid, which
   *          for the purposes of this example is always <code>true</code>.
   */
  private boolean isValidUser(String username, String password) {
    return true;
  }
}
```

11 References

Allen, P. (2001), *Realizing e-Business with Components*, Addison Wesley, ISBN 020167520X

Arsanjani, A. (2002) 'Developing and Integrating Enterprise Components and Services', *Communications of the ACM,* 45(**10**):30-34

Atkinson, S. and Duke, R. (1994) 'A Methodology for Behavioural Retrieval from Class Libraries', *Software Verification Research Centre Technical Report 94-28,* http://citeseer.nj.nec.com/atkinson94methodology.html, last visited: Mar 2003

Atkinson, S. (1997) 'Engineering Software Library Systems', *Software Verification Research Centre,* http://citeseer.nj.nec.com/atkinson97engineering.html, last visited: Mar 2003

Baker, M., Buyya, R. and Laforenza, D. (2002) 'Grids and Grid Technologies for Wide-area Distributed Computing', *Software Practice and Experience,* 32(**15**):1437-1466

Baruchelli, F. and Succi, G. (1997) 'A Fuzzy Approach to Faceted Classification and Retrieval of Reusable Software Components', *ACM Applied Computing Review,* 5(**1**):15-20

Bassett, P.G. (1997) 'Framing Software Reuse: Lessons from the Real World', *Yourdon Press, ISBN: 013327859X*

Belletini, C., Damiani, E. and Fugini, M.G. (2001) 'Software Reuse In-the-Small: Automating Group Rewarding', *Information and Software Technology,* 43(**1**):651-660

Benedicenti, L., Succi, G., Valerio, A. & Vernazza, T. (1996) “Monitoring the Efficiency of a Reuse Program” in *ACM SIGAPP Applied Computing Review,* Vol. 4 issue 2

Bennett, K., Xu, J., Munro, M., Hong, Z., Layzell, P., Gold, N., Budgen, D. and Brereton, P. (2001) 'An Architectural Model for Service-Based Flexible Software', *IEEE COMPSAC*

Bernstein, A. and Klein, M. (2002) 'Towards High-Precision Service Retrieval', *International Semantic Web Conference 2002 (ISWC'02),* http://citeseer.nj.nec.com/596467.html, last visited: Feb 2003

Blair, L., Blair, G.S., Andersen, A. and Jones, T. (2001) 'Formal Support for Dynamic QoS Management in the Development of Open Component-based Distributed Systems', *IEE Proceedings on Software,* 148(**3**):83-92

Bouchachia, A. and Mittermeir, R.T. (2001) 'Coping with Uncertainty in Software Retrieval Systems', *Proceedings of 2nd International Workshop on Soft Computing Applied to Software Engineering (SCASE'01),* http://citeseer.nj.nec.com/473683.html, last visited: Sept 2003

Brereton, O.P. and Budgen, D. (2000) 'Component-Based systems: A Classification of Issues' *IEEE Computer,* 33(**11**):54-62

Brereton, O.P., Linkman, S., Neligwa, T., Bøegh, J. and De Panfilis, S. (2002) 'Software Components – Enabling a Mass Market', *Proceedings of 10th International Workshop on Software Technology and Engineering Practice (STEP 2002), IEEE Computer Society,* http://csdl.computer.org/comp/proceedings/step/2002/1878/00/18780169abs.htm, last visited: Sept 2003

Brown, A.W. (1998) 'From Component Infrastructure to Component-Based Development', *1998 International Workshop on Component-Based Software Engineering (ICSE'98),* http://www.sei.cmu.edu/cbs/icse98/papers/pdf.files/p21.pdf, last visited: Sept 2003

Chen, P.S., Hennicker, R. and Jarke, M. (1993) 'On the Retrieval of Reusable Software Components', *Proceedings of the 2nd International Workshop on Software Use, IEEE Computer,* pp. 99-108

clarifi.eng.it (2003) - CLARiFi Group, http://clarifi.eng.it/, last visited: Oct 2003

Damiani, E. and Fugini, M.G. (1996) 'Design and Code Reuse Based on Fuzzy Classification of Components', *ACM SIGAPP Applied Computing Review, 4*(**2**):26-32

Daminani, E., Fugini, M.F. and Fisaschi, E. (1997) 'A Descriptor-Based Approach to OO Code Reuse', *Feature, IEEE Computer,* pp. 73-80

DCOM, Microsoft Corp. (1996) 'DCOM Technical Overview', http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomtec.asp, last visited: Aug 2003

DCOM, Microsoft Corp. (1997) 'DCOM Architecture', http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomtec.asp, last visited: Aug 2003

De Roure, D., Jennings, N.R. and Shadbolt, N.R. (2003) 'The Semantic Grid: A Future e-Science Infrastructure', http://citeseer.nj.nec.com/context/2059285/0, last visited: Sept 2003

D'Souza, D. (1998) 'Interface Specification, Refinement, and Design with UML/Catalysis', *Journal of Object-Oriented Programming,* 11(**3**):12-18

Duval, E., Forte, E., Cardinaels, K., Verhoeven, B., Van Durm, R., Hendrikx, K., Forte, M.W., Ebel, N., Macowicz, M., Warkentyne, K. and Haenni, F. (2001) 'The ARIADNE Knowledge Pool System' *Communications of the ACM,* 44(**5**):73-78

DeMichiel, L.G., Ümit, L.Y. and Krishnan, S. (2001) 'Enterprise JavaBeans™ Specification 2.0 Final Release 2', http://java.sun.com/webapps/download/Display, last visited: Sept 2003

Di Felice, P. and Fonzi, G. (2002) 'Selecting Reusable Stand-alone Routines: a Proposal and a Case Study', *Journal of Systems and Software,* 54(**3**):171-178

Fischer, B., Kievernagel, M. and Struckmann, W. (1995) 'VCR: A VDM-based Software Component Retrieval Tool', *17th ICSE Workshop on Formal Methods Application in Software Engineering Practice,* http://www.infosun.fmi.uni-passau.de/st/papers/icse17-fmws/, last visited: Aug 2003

Fischer, B. and Snelting, G. (1997) 'Reuse by Contract', *Proceedings of the ESEC/FSE Workshop on Foundations of Component-based Systems,* http://citeseer.nj.nec.com/fischer97reuse.html, last visited: Apr 2003

Fischer, B. (2000) 'Specification-Based Browsing of Software Component Libraries', *Journal of Automated Software Engineering,* 7(**2**):179-200

Foster, I., Kesselman, C., Nick, J.M. and Tuecke, S. (2002) 'Grid Services for Distributed System Integration', *Computing,* 35(**6**):37-46

Fox, C., Danicic, S., Harman, M. and Hierons, R.M. (2003) 'CONSIT: A Fully Automated Conditioned Program Slicer', *Software – Practice and Experience,* pp. 1-32

Frakes, W.B. and Succi, G. (2001) 'An Industrial Study of Reuse, Quality, and Productivity', *Journal of Systems and Software,* 1(**2**):99-106

Franz, M. (1997) 'Dynamic Linking of Software Components', *Feature, IEEE Computer,* 30(**3**):74-81

Fremantle, P., Weerawarana, S. and Khalaf, R. (2001) 'Enterprise Services', *Communications of the ACM,* 45(**10**):77-82

Fuggetta, A., Lavazza, L., Morasca, S., Cinti, S., Oldano, G. & Orazi, E. (Oct 1998) "Applying GQM in an Industrial Software Factory" in *ACM Transactions on Software Engineering and Methodology, Vol. 7 No. 4*

Gomaa, H. and Farrukh, G.A. (1999) 'Methods and Tools for the Automated Configuration of Distributed Applications from Reusable Software Architectures and Components', *IEE Proceedings on Software,* 146(**6**):277-290

Gonzàlez, P.A. (2000) 'Applying Knowledge Modelling and Case-Based Reasoning to Software Reuse', *IEE Proceedings on Software,* 147(**5**):169-177

Grundy, J., Mugridge, W. and Hisking, J. (2000) 'Constructing Component-Based Software Engineering Environments: Issues and Experiences', *Information and Software Technology,* 42(**2**):103-114

Heineman, G.T. and Councill, W. (2001) *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, ISBN: 0201704854

Hemer, D. and Lindsay, P. (2001) 'Specification-based Retrieval Strategies for Module Reuse', *University of Queensland Software Verification Research Centre Technical Report 01-25,* http://svrc.it.uq.edu.au/Publications/2001/svrc2001-025.html, last visited: Feb 2003

Henninger, S. (1997) 'An Evolutionary Approach to Constructing Effective Software Reuse Repositories', *ACM Transactions on Software Engineering Methodology,* 6(**2**):111-140

Jilani, L.L. (1997) 'Retrieving Software Components That Minimize Adaptation Effort', *Automated Software Engineering,* http://citeseer.nj.nec.com/jilani97retrieving.html, last visited: Jan 2003

Kephart, J.O. and Chess, D.M. (2003) 'The Vision of Autonomic Computing', *IEEE Computer,* 36(**1**):41-50

Khayati, O. and Giraudin, J. (2002) 'Components Retrieval Systems', *OOIS Workshop on Reuse in Object Oriented Information Systems Design (OOIS 2002),*

http://www-lsr.imag.fr/OOIS_Reuse_Workshop/Papers/Khayati.pdf, last visited: Aug 2003

Kitchenham, B. (1996) 'Evaluating Software Engineering Methods and Tools, Part 5: The Influence of Human Factors', *ACM Software Engineering Notes,* 22(1):13-15

Kitchenham, B. (1997) 'Evaluating Software Engineering Methods and Tools, Part 7: Planning Feature Analysis Evaluation', *ACM Software Engineering Notes,* 22(4):21-24

Kontogiannis, K., Smith, D. & O-Brien, L. (2003) 'On the Role of Services in Enterprise Application Integration', Proceedings of the 10th International Workshop on Software Technology and Engineering Practice 2002, IEEE Computer, p103

Kreger, H. (2003) 'Fulfilling the Web Services Promise', *Communications of the ACM,* 46(**6**):29-34

Kwon, O.C., Boldyreff, C. and Munro, M. (1997) 'An Integrated Process Model of Software Configuration Management for Reusable Components', *Proceedings of the 9th International Conference on Software Engineering an Knowledge Engineering (SEKE '97),* http://www.dur.ac.uk/computer.science/technical-reports/technical-reports.html, last visited: Apr 2003

Levi, K. and Arsankani, A. (2002) 'A Goal-driven Approach to Enterprise Component Identification and Specification', Communications of the ACM, 45(**10**):45-49

Liao, S.Y., Cheung, L.S. and Liu, W.Y. (1999) 'An Object-Oriented System for the Reuse of Software Design Items', *Journal of Object-oriented Programming,* 11(**8**):22-28

Maletic, J.I. and Marcus, A. (2001) 'Supporting Program Comprehension Using Semantic and Structural Information', *Proceedings of the 23rd IEEE International Conference on Software Engineering (ICSE'2001),* http://citeseer.nj.nec.com/maletic01supporting.html, last visited: Feb 2003

McArthur, K., Saiedian, H. and Zand, M. (2002) 'An Evaluation of the Impact of Component-Based Architectures on Software Reusability', *Information and Systems Technology,* 44(**6**):351-359

Menu, M. (2001) 'Achieving Plug-and-Play Behavior within a Class', *Journal of Object-oriented Programming,* 13(**9**):18-22

Mili, H., Mili, F. and Mili, A. (1995) 'Reusing Software: Issues and Research Directions', *IEEE Transactions on Software Engineering,* 21(**6**):528-62

Mittermeir, R.T. and Pozewaunig, H. (1998) 'Classifying Components by Behavioral Abstraction', *Proceedings of the 4th Joint Conference on Information Sciences (JCIS'98),* http://citeseer.nj.nec.com/mittermeir98classifying.html, last visited: Mar 2003

Mittermeir, R.T. & Pozewaunig, H. (2002) 'Self-descriptive Software Components', *16th European Meeting on Cybernetics & Systems Research (EMCSR'2002),* http://citeseer.nj.nec.com/598440.html, last visited: Sept 2003

Nova Laboratories (1999) 'The Developer's Guide to Understanding Enterprise JavaBeans Applications', *Nova Laboratories,* http://www.nova-labs.com, last visited: Sept 2003

OPEN Group (1999) 'The ActiveX Core Technology Reference', http://www.opengroup.org/onlinepubs/009899899/toc.htm, last visited: Sept 2003

Parrish, A., Dixon, B. and Cordes, D. (2001) 'A Conceptual Foundation for Component-Based Software Deployment', *Journal of Systems and Software,* 57(**3**):193 - 200

Penix, J. and Alexander, P. (1997) 'Component Reuse and Adaptation at the Specification Level', *Proceedings of the 8th Annual Workshop on Software Reuse (WSIR '97),* http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/penix/penix.html, last visited: Oct 2003

Penix, J. and Alexander, P. (1999) 'Efficient Specification-Based Component Retrieval', *Automated Software Engineering,* 6(**2**):139-170

Pressman, R. S. (1997) "Software Engineering: A Practitioner's Approach" (4th Ed.)

Prieto-Díaz, R. (1995) 'Implementing Faceted Classification for Software Reuse', *Communications of the ACM,* 34(**5**):88-97

Rastofer, U. and Bellosa, F. (2001) 'Component-Based Software Engineering for Distributed Embedded Real-time Systems', *IEE Proceedings on Software,* 148(**3**):99-103

Ravichandran, T. and Rothenberger, M.A. (2003) 'Costware Reuse Strategies and Component Markets', *Communications of the ACM,* 46(**8**):109-114

Rine, D.C. and Nada, N. (1999) 'An Empirical Study of a Software Reuse Reference Model', *Information and Software Technology,* 42(**1**):47-65

Rodden, K. and Blackwell, A. (2002) 'Class Libraries: A Challenge for Programming Usability Research', *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group (PPIG),* http://www.ppig.org/papers/14th-rodden.pdf, last visited: Dec 2002

Saar, R. (2000) 'Extension of Software Components using Multimethods', *Journal of Object-Oriented Programming,* 13(**1**):12-16

Saleh, K. and Al-Saqabi, K. (1998) 'Error Detection and Diagnosis for Fault Tolerance in Distributed Systems', *Journal of Information and Software Technolog*y, 39(**14**):975-983

Schumann, J. and Fischer, B. (1997) 'NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical', *Proceedings of Automated Software Engineering, IEEE Computer Society Press,* http://citeseer.nj.nec.com/432304.html, last visited: Mar 2003

Sousa, J.P. and Garlan, D. (2001) 'Formal Modeling of the Enterprise JavaBeans Component Integration Framework', *Information Systems Technology,* 43(**3**):171-188

Stal, M. (2002) 'Web Services: Beyond Component-Based Computing', *Communications of the ACM,* 45(**10**):71-76

Sutcliffe, A. (2000) 'Domain Analysis for Software Reuse', *Journal of Systems and Software,* 50(**3**):175-199

Sutherland, J. and van den Heuvel, W. (2002) 'Enterprise Application Integration and Complex Adaptive Systems', *Communications of the ACM,* 45(**10**):59-64

Szyperski, C. (1998) 'Component Software: Beyond Object-Oriented Programming', *Addison-Wesley, ISBN: 0201178885*

Talbert, N. (1998) 'The Cost of COTS', *Computing,* 31(**6**):46-52

Thomas, A. (1998) 'Enterprise JavaBeans Technology: Server Component Model for the Java Platform', *Sun Microsystems White Paper,* http://java.sun.com/products/ejb/white_paper.html, last visited: Aug 2003

Thomas, A. (2003) 'Enterprise JavaBeans: Server component model for Java', Sun Microsystems White Paper, http://java.sun.com/products/ejb/white_paper.html, last visited: Aug 2003

Thomason, S. and Brereton, O.P. (2002) 'Supporting Evolution and Maintenance of Components using a Remote Service Architecture', *IEEE COMPSAC*

Tilley, T., Cole, R., Becker, P. and Eklund, P. (2003) 'A Survey of Formal Concept Analysis Support for Software Engineering Activities', *Proceedings of 1st International Conference on Formal Concept Analysis (ICFCA'03),* http://www.kvocentral.com/kvopapers/tilley03survey.ps.gz, last visited: Aug 2003

Torchiano, M., Jaccheri, L., Sørensen, C-F. and Wang, A.I. (2002) 'COTS Products Characterization', *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02),* http://www.ifi.uio.no/~isu/INCO/Papers/Cots_SEKE2002.pdf, last visited: Aug 2003

Turner, M., Budgen D. and Brereton, P. (2003) 'Turning Software into a Service', *IEEE Computer,* 36(**10**):38-44

Vitharana, P., Zahedi, F. and Jain, H. (2003) 'Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis', *IEEE Transactions on Software Engineering,* 29(**7**):649-664

Chinnici, R., Gudgin, M., Moreau, J. and Weerawarana, S. (2003) 'Web Services Description Language (WSDL) Version 1.2 (W3C Working Draft 3)', http://www.w3.org/TR/2003/WD-wsdl12-20030303, last visited: Sept 2003

Ye, Y. and Fischer, G. (2001) 'Context-Aware Browsing of Large Component Repositories', *IEEE 16th International Conference on Automated Software Engineering (ASE'01),* http://citeseer.nj.nec.com/ye01contextaware.html, last visited: July 2003

Ye, Y. and Fischer, G. (2002) 'Supporting Reuse by Delivering Task-Relevant and Personalized Information', *Proceedings of 2002 International Conference on Software Engineering (ICSE'02),* http://citeseer.nj.nec.com/465022.html, last visited: May 2003

Zaremski, A.M. (1995) 'Signature Matching: a Tool for Using Software Libraries', *ACM Transactions on Software Engineering and Methodology,* 4(**2**):146-170

Zaremski, A.M. (1996) 'Signature and Specification Matching', *PhD thesis,* http://citeseer.nj.nec.com/162469.html, last visited: May 2003

Zaremski, A.M. and Wing, J.M. (1997) 'Specification Matching of Software Components', *ACM Transactions on Software Engineering and Methodology,* 6(**4**):333-369

Zarras, A. and Issarny, V. (1998) 'Imposing Transactional Properties on Distributed Software Architectures', *Proceedings of the 8th ACM SIGSOFT International Software Architecture Workshop*, http://www.acm.org/sigs/sigops/EW98/papers.html, last visited: Sept 2003