MANCHESTER
1824
The University
of Manchester

Computer Science
University of Manchester

Deployment Contracts for Software Components

Kung-Kiu Lau and Vladyslav Ukis

Department of Computer Science
University of Manchester
Preprint Series
CSPP-36

Deployment Contracts for Software Components

Kung-Kiu Lau and Vladyslav Ukis

February 2006

Abstract

Today's approaches to software architecture mostly regard components to be black boxes with interfaces. A component's interface is made up of signatures of operations offered by the component. The interface is the only entity visible outside of component. This means that component composition can be done only at the interface level since component's interiors are hidden in the black box. Although composition at the interface level empowers component integrators to compose applications out of preexisting components, the information visible outside component is not sufficient to reason about component composition [53]. In other words, compositional reasoning is hardly supported by current approaches to software architecture. Furthermore, predictable assembly of components is currently not supported.

In this report, we present an approach to augment component's interface with metadata. The metadata describes component's behaviour. In particular, our metadata set is comprised of attributes describing component's environmental dependencies as well as threading model. The metadata is analysed to allow compositional reasoning of components. Moreover, we consider two widely-spread execution environments of components, desktop and web environment. We use the metadata to prevent components' conflicts with the target execution environment. Furthermore, we analyse the metadata at component deployment time to discover conflicts due to happen at runtime. That is, we propose an approach where conflicts arising from incompatible component's properties with the properties of the target execution environment and the properties of other components can be statically predicted before runtime.

Keywords: components, component models, component contracts, component metadata, deployment contracts, component composition, .NET, J2EE

Copyright © 2000, University of Manchester. All rights reserved. Reproduction (electronically or by other means) of all or part of this work is permitted for educational or research purposes only, on condition that no commercial gain is involved.

Recent preprints issued by the Department of Computer Science, Manchester University, are available on WWW via URL `http://www.cs.man.ac.uk/preprints/index.html` or by ftp from `ftp.cs.man.ac.uk` in the directory `pub/preprints`.

Contents

1 Introduction **5**

2 Problem Domain **5**
2.1 Software Components 6
2.1.1 Definition 6
2.1.2 Lifecycle Phases of Software Components 6
2.2 Component Metadata 8

3 Comprehensive Frameworks for Component Development **10**
3.1 Overview 10
3.2 J2EE Framework 11
3.3 .NET Framework 16
3.4 Summary 22

4 Deployment Contracts for Components **22**
4.1 Component Dependencies 22
4.1.1 Possible General Component Dependencies 23
4.1.2 Application of General Component Dependencies to JavaBeans 26
4.1.3 Application of General Component Dependencies to EJB 27
4.1.4 Application of General Component Dependencies to .NET Component Model 28
4.2 Attributes for Components 29
4.2.1 Compiler input and output files 32
4.2.2 Entries in a registry 32
4.2.3 Dynamically activated types 33
4.2.4 Framework Store 34
4.2.5 Configuration and Licensing of the execution environment 35
4.2.6 Environment variables 35
4.2.7 Database connections 36
4.2.8 Database tables in relational databases 37
4.2.9 Database Connectivity 37
4.2.10 Performance counter 38
4.2.11 Log Files 38
4.2.12 Event Logs 39
4.2.13 Processes 40
4.2.14 Preferred Processor 40
4.2.15 Directory Services 41
4.2.16 Files containing graphics 41
4.2.17 Fonts 42
4.2.18 Printers 42
4.2.19 COM components 42
4.2.20 SOAP components on web servers 44
4.2.21 String culture 44
4.2.22 File system 45
4.2.23 Message queues 46

4.2.24 Network-specific dependencies . . . 46
4.2.25 Reflected methods or properties of a type . . . 47
4.2.26 Language resources . . . 48
4.2.27 Remote method invocations . . . 48
4.2.28 Communication channels . . . 49
4.2.29 Cryptography certificate files . . . 50
4.2.30 Residential Services . . . 51
4.2.31 Character encoding . . . 51
4.2.32 Threading Model . . . 52
4.2.33 Thread-specific storage . . . 56
4.2.34 Web applications' dependencies . . . 57
4.2.35 Web services-related dependencies . . . 60
4.2.36 Advertisement files . . . 61
4.2.37 Cursors . . . 61
4.2.38 Icons . . . 62
4.2.39 XML-related dependencies . . . 62
4.2.40 Audio and video content . . . 63
4.2.41 Colour profile . . . 64
4.2.42 Email-related dependencies . . . 64
4.2.43 Hardware communication ports . . . 66
4.2.44 Other attributes . . . 66
4.3 Examples of Deployment Contracts . . . 68

5 System Execution Environments **70**
5.1 Desktop Environment . . . 70
5.2 Web Environment . . . 71
5.3 Resource Availability . . . 76
5.4 Summary . . . 77

6 System-specific Properties **79**
6.1 System State . . . 80
6.2 System Threading Model . . . 80
6.3 Summary . . . 80

7 Framework for Deployment Contracts Analysis **81**
7.1 Example of Deployment Contract Analysis . . . 84

8 Evaluation **85**

9 Conclusion **88**

List of Figures

1 Component lifecycle phases . . . 7
2 J2EE Overview . . . 11
3 .NET Overview . . . 11
4 J2EE Class Library at a Glance . . . 12
5 High-level view of .NET Framework functionality . . . 17
6 .NET Class Library at a Glance . . . 18
7 Possible General Component Dependencies . . . 23
8 Component dependencies in JavaBeans . . . 27
9 Component dependencies in EJB . . . 28
10 Component dependencies in .NET Component Model . . . 29
11 A component with an environmental dependency. . . . 68
12 A component with a defined threading model. . . . 69
13 A component with a defined threading model. . . . 69
14 A component with a defined threading model. . . . 69
15 Properties (or strengths) to be resolved in a component-based system development 82
16 Overview of the proposed component model . . . 83
17 Implementation of a design pattern for components with use of metadata attributes 84

List of Tables

1 Component models utilizing component metadata 9
2 System instantiation modes in the web environment and their effects 75
3 Properties of the desktop and web execution environments 78
4 Resources available to components in an execution environment 79
5 Considered System-specific properties 81
6 Defined properties for a component-based system development 83
7 A comparison of component models utilizing component metadata 87

1 Introduction

In recent years, many approaches to software architecture [40, 4, 39, 23, 8, 9, 49, 50] have been developed and utilized. They include software component models and architecture description languages (ADLs). They all have in common that they define components and composition rules among them. That is, a component is a different entity in nearly each of current approaches. Composition rules also differ among component models and ADLs. In this work we primarily look into components. Components in current approaches to software architecture are largely black box entities with an interface. The component's interface shows off operations offered by the component. It is used for component composition and relieves the application developer of having to look inside component. Thus, an application should be composable of components by only considering their interfaces. However, problems have been reported with this approach [53]. Today's component interfaces do not disclose any component behaviour. This leads to a difficult situation for the application developer in which they have to choose components for their systems knowing only syntax of operations offered by the component. Thus, the application developer can only discover the behaviour of the component at runtime after they have integrated it into their system. This is certainly insufficient and desirable to be able to analyse component behaviour before even choosing the component for a system. In this work, we propose to augment the component with metadata, which is, like component's interface, visible outside component. The metadata can be retrieved from the component before runtime and analysed. Moreover, we show that we can do compositional reasoning by checking the meatadata attached to composed components for mutual compatibility. We thus can predict conflicts arising from incompatible properties of composed components.

Furthermore, we define properties of today's widely used component execution environments, desktop and web environment, and show that we can use the metadata attached to the component for predicting conflicts arising from incompatible properties of the component and its target execution environment.

Finally, we introduce a new concept in this work that we call 'Deployment Contract' of component. A Deployment Contract of component is a set of metadata attached to the component showing its runtime behaviour that can be retrieved and analysed at component deployment time. We present a pool of metadata attributes we have developed to be attached to components. The attributes are in the suitable shape to be retrieved at component deployment time, i.e. before component instances are created. They are furthermore highly parameterisable, which makes them particularly suitable for disguising component runtime behaviour.

2 Problem Domain

There is little unified terminology in the area of software architecture [25]. The term 'software component' is not uniformly defined. Neither are software component's lifecycle phases. In this section we introduce these notions, which are fundamental and used throughout the report. It is important to make clear what is understood under the term 'software component' here and to define software component's lifecycle phases before proceeding. Furthermore, we introduce the notion of component metadata to map out the problem domain of this report.

2.1 Software Components

In this section we define what we understand under the term software component as well as define software component's lifecycle phases.

2.1.1 Definition

The term Software Component is not uniformly defined in Component-Based Software Development (CBSD) and has been used in the literature in various meanings. Therefore it is essential to clearly state what is understood by Software Component in this work before proceeding. There are several currently most adopted definitions of software component today. Among them there are two, a mixture of which this work will adopt, with some amendments.

First definition is given by Szyperski [62] and is the following:

```
''A software component is a unit of composition with contractually
specified interfaces and explicit context dependencies only. A
software component can be deployed independently and is subject to
composition by third parties.''
```

The second one is given by Heineman and Council [62] and states that

```
''A [component is a] software element that conforms to a component
model and can be independently deployed and composed without
modification according to a composition standard.''
```

We think that on the one hand the definition given by Szyperski lacks the fact that a component[1] must conform to a component model and on the other hand the definition proposed by Heineman and Council does not mention that components interface is a contract between component and its clients. Both definitions adopt the fact that a component is unit of independent deployment, which we fully agree with.

None of the definitions states clearly, however, that a component should have a deployment contract for describing its runtime behaviour, which we believe to be an important characteristic of component. A deployment contract tells component client at component deployment time (i.e. after compilation and before instantiation) how the component will behave at runtime. Szyperski's definition states that a component is a unit with contractually specified interfaces. Nonetheless, it does not argue that the contract should comprise component's runtime behaviour. It states that component's context dependencies should be explicit but does not say that those dependencies should also be part of component's contract.

2.1.2 Lifecycle Phases of Software Components

Having clarified what we understand under the term 'component' we go on to describe what we think to be phases in component's lifecycle as those are generally not clearly defined in CBSD.

We identify 6 phases a component can go through during its lifecycle. These are 'Design Phase', 'Compilation Phase', 'Deployment Phase', 'Runtime Phase', 'Update Phase' and 'Removal Phase'. The relationships between the phases are depicted in Figure 1 and explained in the following.

[1] We mean 'software component' when referring to the term 'component' in this work.

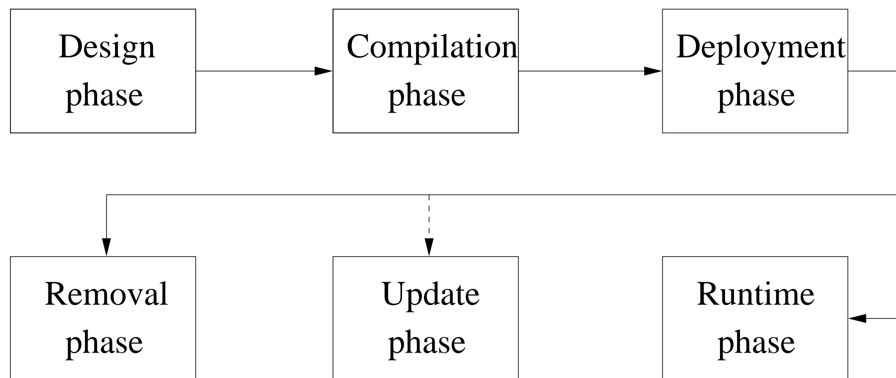

Figure 1: Component lifecycle phases

Component design phase In the component design phase, software component is designed. This includes all activities which lead to component's source code in a programming language ready to be compiled. During design phase a component can be specified using some specification (e.g. UML or an ADL specification), which will ultimately be compiled into source code. So, the input of design phase of component is an idea about what component is supposed to do and the output is complete source code for the component. In design phase composite components can be built from component templates. But the outcome remains the same. At the end of the component design phase, a component is designed and implemented, i.e. its source code is available and ready to be compiled.

Component compilation phase This phase takes as input the source code from the design phase and compiles the component into a binary unit using a compiler for the programming language the component is implemented in. This step is completely automated. Component developer is not involved in compilation further than to start it off. The outcome of the component compilation phase is a binary component, which is ready to be deployed. Binary components are ready to be sold as they, unlike the source code from component design phase, do not reveal intellectual property about component implementation in a human readable form.

Component deployment phase In this phase a binary component can be either integrated into an application or become part of a composite component (in fact a component is intended to be integrated into as many applications or composite components as possible to foster reuse). The application itself the component is being integrated into is in its design phase, but the component is in deployment phase. This is an important fact, which often causes confusion of terminology. The same applies when a binary component is composed with another component to build a composite component. The composite component is being designed and making use of the binary component, which is in deployment phase.

An important distinction between component design and deployment phase is that at design phase component's code is designed, developed and changed, whereas at deployment phase the code is not changed any more and the component is in binary form ready for further integration or composition. It is worth stressing that the system the component is being integrated to is in its design phase whereas the component itself is already in deployment phase.

At deployment phase, various tasks may be necessary to prepare the component to run. For instance, a component may need to be registered with a naming service to be looked up at runtime. Some component models employ containers for hosting components. Components are required to be provided with some descriptors to allow containers to instantiate them at runtime.

To summarise, the input of component deployment phase is a binary component and the output is the binary component prepared to execute in the target environment. I.e. the component is integrated into the target environment be it an application or a composite component and is registered with some system entities like naming services or containers if necessary.

Component runtime phase At component runtime phase a component instance is created and running. The input for the runtime phase of component is a deployed component and the output is a component instance, which is expected to provide its clients with provided services when its required services are satisfied. The composition of components with fitting provided and required services resulting in a system is done at system design time when components are in deployment time. This composition cannot be changed any more at runtime (unless system runtime reconfiguration is supported like in [38], which is not the case in general). Therefore, it is important to carefully select suitable components at system design time. As components are in deployment phase at system design time, it is essential for components to have a neat deployment contract to enable system designer to reasonably select components for their system. As shown above todays component definitions do not mention deployment contracts, which we consider essential.

Component update phase In this phase component is updated with a new version of it. This phase is optional, which is illustrated in Figure 1 by a dotted line. If a system is not updated, its components are not updated either. The input for this phase is the old deployed component with a new component to be deployed. The output of the phase is the new deployed component. In this phase, the new component has to be deployed to the system, i.e. the steps necessary to deploy a component have to be repeated. Those can include reregistering a new version of component with a naming service or providing a new component descriptor for the component container.

Component removal phase At this phase the component gets removed from the system. The steps that were necessary to deploy the component have to be reversed. Thus the input for this phase is a deployed component and the output is the vanishing of the component from the system.

Now that, we have defined all the basic concepts for our research and go on to expand on component metadata in the next section.

2.2 Component Metadata

Metadata in general is data describing data. The concept of metadata has been used throughout many areas in Computer Science. Database Systems rely on metadata to efficiently manage data stored in database management systems. Web services expose metadata to be discovered by web clients. Metadata is used in hardware systems to specify behaviour of their system parts. In the component-based software development, however, the use of metadata to specify behaviour

of software components has not proliferated yet. In general, todays component models use interfaces to show the client how to use a component, which is considered a black box. The interface exposes syntactical contract of the component, i.e. method and event signatures offered by the component, but does not reveal any behavioural aspects inside the black box. This makes it difficult to reason about component's behaviour before integrating it into a system and impossible to differentiate between components possessing the same (syntactical) interface. Only few of the current component models use metadata to expose some component behaviour. And if at all then component metadata is intended to be used not by the component client but by the component's host, a container.

In the previous sections we identified the need for components to have deployment contracts analysed at component deployment time. A Deployment Contract is metadata about a software component describing its behaviour. An additional essential characteristic of a deployment contract is that it is analysed at deployment time of component (i.e. before runtime instance is created). In [29] we surveyd component models utilizing component metadata and showed that none of today's approaches can analyse component metadata at component deployment time and perform compositional reasoning. We briefly show that by comparison of an extension of ACME ADL presented in [57], work presented in [48] on lightweight metadata-based extensions of Component Models, Enterprise Java Beans (EJB) [42, 23, 13, 44, 63], CORBA Component Model (CCM) [51, 67, 49] and .NET [12, 45] in Table 1.

Property	ACME Ext.	Comp. Model Ext.	EJB	CCM	.NET
Metadata processed at component deployment	No	No	No	No	Yes
Metadata used for component composition analysis	Yes	No	No	No	No

Table 1: Component models utilizing component metadata

The comparison shows clearly that none of the systems from Table 1 can perform compositional reasoning of components at their deployment time. .NET Component Model can process metadata about component rendering by an application designer tool at component deployment time. Such analysis does not take into account component behaviour. In other words, component behaviour is not analysed by current approaches at component deployment time [29]. ACME Extension uses metadata to perform component composition analysis of inter-component pathways.

Our problem domain in this work is to develop a pool of attributes, embodying component metadata, which can be attached to components (of the kind defined in Section 2.1) and analysed on their deployment. A set of those attributes attached to a component is its Deployment Contract. We concentrate on attributes related to component's environmental dependencies and threading model. In other words, using the attributes a component will be able to show off its environmental dependencies as well as threading model. Furthermore, we want to develop a framework for analysing the deployment contracts of components at their deployment time to predict runtime conflicts.

Moreover, we aim at defining properties of two widely used execution environments, desktop and web environment, and checking deployment contracts of components for compatibility with the target execution environment.

To develop attributes related to component's environmental dependencies and threading model as well as define properties of the desktop and web environment, we choose to analyse

general-purpose frameworks for software development. The frameworks allow building components by utilizing rich functionality inside them. Furthermore, they allow software development for desktop and web execution environment making themselves particularly suitable analysis candidates for our endevaour.

3 Comprehensive Frameworks for Component Development

Today there are two general comprehensive frameworks that allow application software development [18, 22, 66]: Java 2 Enterprise Edition (J2EE) and .NET Framework. Using J2EE and .NET it is possible to develop software applications of nearly any kind. In the following sections we give an overview of the two frameworks.

3.1 Overview

Figures 2 and 3 show the overall picture of J2EE and .NET respectively.

The symmetry of the frameworks can be recognised by comparing the figures. Using either of the frameworks complex object-oriented software systems can be implemented. J2EE Technology is more mature than .NET, whereas .NET is more modern.

Furthermore, both frameworks abstract from operating system. That is, systems implemented (and compiled) on one operating system can run on any other one supporting the runtime of the framework. This is achieved by compiling the source code not into byte code of a particular platform like e.g. in C++ but rather into intermediate byte representation. The intermediate byte representation gets compiled into actual byte code of the platform the code is executing on as early as the code is loaded by the runtime. A specific runtime is provided for each platform that can compile intermediate byte code into the actual byte code of the platform it is for.

Moreover, both frameworks allow building systems for desktop or web deployment. When building systems for desktop deployment, the system must be installed on a computer before it can be used. The user usually interacts with the system via its user interface. If the user interface is not available, the user usually interacts with the system via command shell. When building systems for the web deployment, the system is deployed to a web server and runs under its control. The user usually accesses a system deployed to web server via their web browser. In addition, not only web browsers but also devices like Personal Digital Assistants (PDA) or mobile phones can access system deployed to web server.

Additionally, both frameworks offer subsystems for accessing databases.

Also, both frameworks offer functionality for security management in applications as well as transaction management and concurrency.

Furthermore, both frameworks offer class libraries implementing various protocols for accessing remote objects.

In addition, both frameworks introduce the notion of context. An object can be logically assigned to a context. An object assigned to a context is called context-bound object. The frameworks allow interception of calls if an object bound to one context invokes another object bound to another object. In other words, the frameworks allow interception of invocations on crossing context boundaries. Hooks can be provided that will be executed on cross-context invocations.

All in all, from the conceptual point of view both J2EE and .NET framework offer nearly the same functionality. Implementations of the concepts differ, of course.

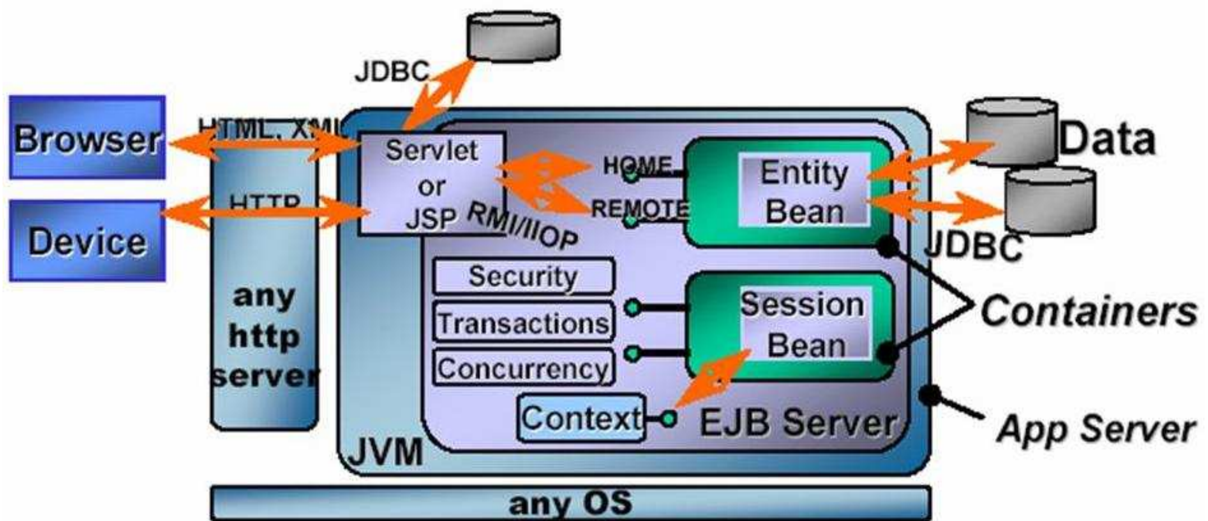

Figure 2: J2EE Overview

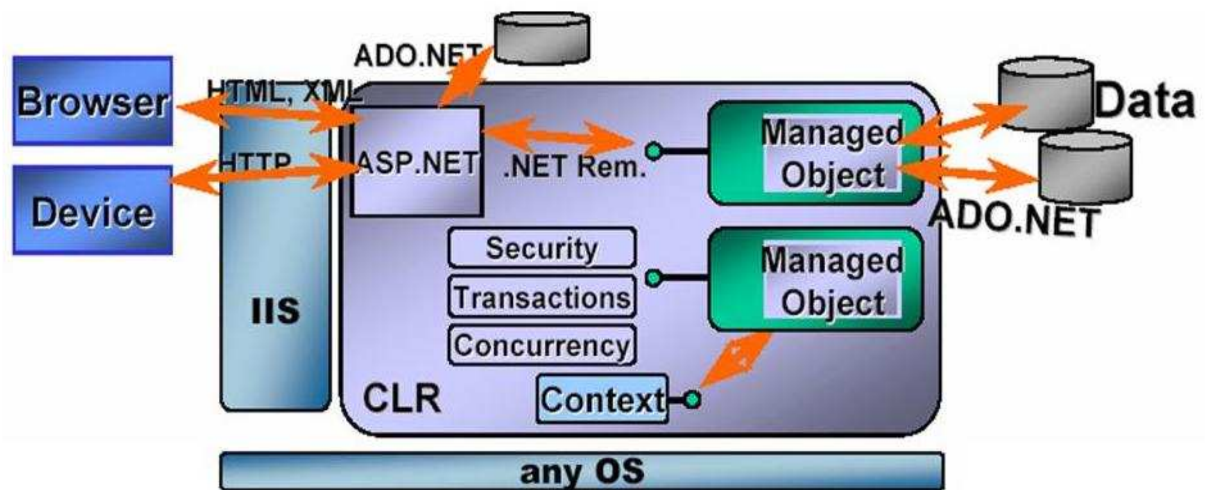

Figure 3: .NET Overview

In this section, we introduced the J2EE and .NET frameworks with a high-level description. In the next section, we go in-depth and present each framework more detailed.

3.2 J2EE Framework

J2EE Framework [54, 59, 37] illustrated in Figure 2 is based on Java Virtual Machine (JVM). JVM allows Java programs to run on any platform that has a JVM implementation. The JVM of a platform compiles intermediate byte code of java binaries produced by Java compiler into the actual byte code of the platform.

For web applications, J2EE offers Java Server Pages (JSP) or Servlet technology (Figure 2). A servlet is a special object that lives in a special servlet container. A servlet can access

database resources using Java Database Connectivity (JDBC). It can furthermore access entity or session beans, which are also special objects that live in special Enterprise Java Beans (EJB) containers.

Additionally, entity beans can in turn access database resources by using JDBC (Figure 2).

Furthermore, security services, transaction handling and concurrency as well as contexts for objects are offered to J2EE applications.

We need to take a closer look at J2EE to be able to find out subsystems using which a component can incur environmental dependencies or influence its threading model. In other words, we need to get down to J2EE's framework class library illustrated in Figure 4.

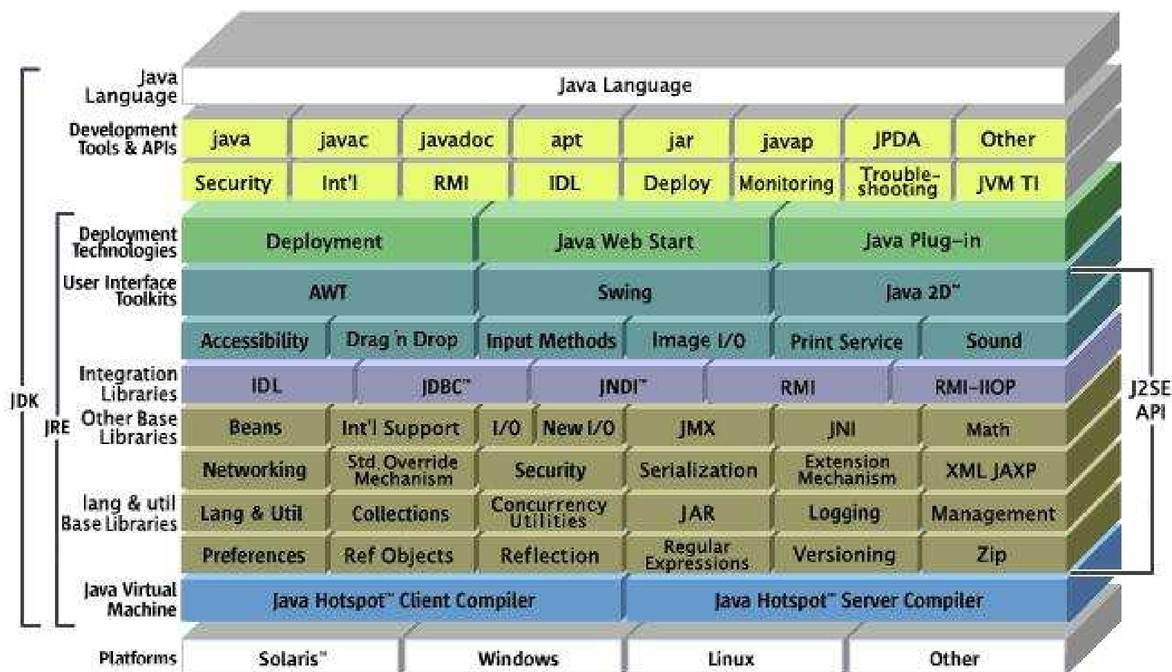

Figure 4: J2EE Class Library at a Glance

??? Figure 4 shows various subsystems comprising J2EE's functionality. The subsystems are ready-to-use by components being developed on the basis of the J2EE framework.

At the bottom in Figure 4 platforms supporting Java are depicted. They include Solaris, Windows, Linux etc. That is, J2EE is a platform-independent framework since it uses a platform-independent programming language, Java.

Further up in Figure 4 Java Virtual Machine (JVM) is shown. A JVM is always platform-dependent. There are two versions of JVM: client and server JVM. The client JVM is typically used for client applications. It is tuned for reducing start-up time and memory footprint. The server JVM is designed for maximum program execution speed. A JVM can be started it client or server mode depending on application's needs.

Further up in Figure 4 libraries providing basic features and fundamental functionality for the Java platform are illustrated. Among them Lang and Util Packages provide rich basic functionality including:

Collections Framework A collection is an object that represents a group of objects. The

collections framework is a unified architecture for representing collections, allowing them to be manipulated independently of the details of their representation.

Concurrency Utilities The Concurrency Utilities packages provide a powerful, extensible framework of high-performance threading utilities such as thread pools and blocking queues. Additionally, these packages provide low-level primitives for advanced concurrent programming.

Java Archive (JAR) Files JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the applet author can digitally sign individual entries in a JAR file to authenticate their origin.

Logging The Logging APIs facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams. The Logging APIs capture information such as security failures, configuration errors, performance bottlenecks, and/or bugs in the application or platform.

Monitoring and Management Comprehensive monitoring and management support for Java platform including Monitoring and Management API for Java virtual machine, Monitoring and Management API for the Logging Facility, Java Management Extensions (JMX) etc.

Preferences The Preferences API provides a way for applications to store and retrieve user and system preference and configuration data. The data is stored persistently in an implementation-dependent backing store. There are two separate trees of preference nodes, one for user preferences and one for system preferences.

Reference Objects Reference objects support a limited degree of interaction with the garbage collector. A program may use a reference object to maintain a reference to some other object in such a way that the latter object may still be reclaimed by the collector. A program may also arrange to be notified some time after the collector has determined that the reachability of a given object has changed. Reference objects are therefore useful for building simple caches as well as caches that are flushed when memory runs low, for implementing mappings that do not prevent their keys (or values) from being reclaimed, and for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.

Reflection Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions. The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.

Package Version Identification The package versioning feature enables package-level version control so that applications and applets can identify at runtime the version of a specific Java Runtime Environment, VM, and class package.

Other Base Packages shown further up in Figure 4 include:

JavaBeans^{TM} Component API Contains classes related to developing beans - components based on the JavaBeans^{TM} architecture that can be pieced together as part of developing an application.

Internationalization APIs that enable the development of internationalized applications. Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes.

I/O Input/Outpur (I/O) functionality provides for system input and output through data streams, serialization and the file system.

New I/O The new I/O (NIO) APIs provide new features and improved performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching.

Java Mangement Extensions (JMX) The Java Management Extensions (JMX) API is a standard API for management and monitoring of resources such as applications, devices, services, and the Java virtual machine. Typical uses include consulting and changing application configuration, accumulating statistics about application behavior, and notifying of state changes and erroneous conditions. The JMX API includes remote access, so a remote management program can interact with a running application for these purposes.

Java Native Interface (JNI) Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications. The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform.

Math Math functionality includes floating point libraries and arbitrary-precision math.

Networking Provides classes for networking functionality, including addressing, classes for using URLs and URIs, socket classes for connecting to servers, networking security functionality, and more.

Security APIs for security-related functionality such as configurable access control, digital signing, authentication and authorization, cryptography, secure Internet communication, and more.

Object Serialization Object Serialization extends the core Java Input/Output classes with support for objects. Object Serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI).

XML (JAXP) A rich set of APIs for processing XML documents and data.

Further up in Figure 4 Integration Libraries are depicted. They include:

Java IDL (CORBA) Java IDL technology adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Java IDL enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group. Runtime components include a Java ORB for distributed computing using IIOP communication.

Java Database Connectivity (JDBC) API The JDBC^{TM} API provides universal data access from the Java programming language. Using the JDBC developers can write applications that can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which tools and alternate interfaces can be built.

Java Naming and Directory Interface^{TM} (JNDI) API The Java Naming and Directory Interface^{TM} (JNDI) provides naming and directory functionality to applications written in the Java programming language. It is designed to be independent of any specific naming or directory service implementation. Thus a variety of services–new, emerging, and already deployed ones–can be accessed in a common way. The JNDI architecture consists of a API and an SPI (Service Provider Interface). Java applications use this API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, allowing the Java application using the JNDI API to access their services.

Remote Method Invocation (RMI) Remote Method Invocation (RMI) enables the development of distributed applications by providing for remote communication between programs written in the Java programming language. RMI enables an object running in one Java Virtual Machine to invoke methods on an object running in another Java VM, which may be on a different host.

RMI-IIOP Java Remote Method Invocation over Internet Inter-ORB Protocol technology The RMI Programming Model enables the programming of CORBA servers and applications via the RMI API. You can choose to work completely within the Java programming language using the Java Remote Method Protocol (JRMP) as the transport, or work with other CORBA-compliant programming languages using the Internet InterORB Protocol (IIOP).

Further up in Figure 4 User Interface Toolkits are depicted. They include:

AWT The Java^{TM} platform's Abstract Windowing Toolkit (AWT) provides APIs for constructing user interface components such as menus, buttons, text fields, dialog boxes, checkboxes, and for handling user input through those components. In addition, AWT allows for rendering of simple shapes such as ovals and polygons and enables developers to control the user-interface layout and fonts used by their applications.

Swing The Swing APIs also provide graphical component (GUI) for use in user interfaces. The Swing APIs are written in the Java programming language without any reliance on code that is specific to the GUI facilities provided by underlying operating system. This allows the Swing GUI components to have a "pluggable" look-and-feel that can be switched while an application is running.

Java 2D^{TM} Graphics and Imaging The Java 2D^{TM} API is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators.

Accessibility With the Java Accessibility API, developers can easily create Java applications that are accessible to disabled persons. Accessible Java applications are compatible with assistive technologies such as screen readers, speech recognition systems, and refreshable braille displays.

Drag and Drop Data Transfer Drag and Drop enables data transfer both across Java programming language and native applications, between Java programming language applications, and within a single Java programming language application.

Input Method Framework The input method framework enables the collaboration between text editing components and input methods in entering text. Input methods are software components that let the user enter text in ways other than simple typing on a keyboard. They are commonly used to enter Japanese, Chinese, or Korean – languages using thousands of different characters - on keyboards with far fewer keys. However, the framework also supports input methods for other languages and the use of entirely different input mechanisms, such as handwriting or speech recognition.

Image I/O The Java Image I/O API provides a pluggable architecture for working with images stored in files and accessed across the network. The API provides provides a framework for the addition of format-specific plugins. Plug-ins for several common formats are included with Java Image I/O, but third parties can use this API to create their own plugins to handle special formats.

Print Service The Java^{TM} Print Service API, allows printing on all Java platforms.

Sound The Java platform includes a powerful API for capturing, processing, and playing back audio and MIDI (Musical Intstrument Digital Interface) data. This API is supported by an efficient sound engine which guarantees high-quality audio mixing and MIDI synthesis capabilities for the platform.

Above User Interface Toolkits in Figure 4 there are deployment and development tools, which we do not explain here.

In the next section we go on to elaborate on the second comprehensive framework for software development, .NET Framework.

3.3 .NET Framework

.NET Framework [65, 12, 36, 41, 52] illustrated in Figure 3 is based on Common Language Runtime (CLR). CLR allows .NET programs to run on any platform that has a CLR implementation. The CLR of a platform compiles intermediate byte code of .NET binaries produced by a .NET compiler into the actual byte code of the platform. The efforts for CLR implementations for various platforms are ongoing in Mono project [43]. There is already a CLR for Linux, Solaris and Mac platform.

For web applications, .NET offers Active Server Pages (ASP.NET) technology (Figure 3). An ASP.NET Page is a special object that lives on a web server and is managed by the ASP.NET runtime. ASP.NET objects can access database resources using a database connectivity (ADO.NET). They can furthermore access other managed objects, which are ordinary objects that are managed by CLR.

Additionally, managed objects can in turn access database resources by using ADO.NET (Figure 3).

Furthermore, security services, transaction handling and concurrency as well as contexts for objects are offered to .NET applications.

We need to take a closer look at .NET to be able to find out subsystems using which a component can incur environmental dependencies or influence its threading model. In other words, we need to get down to .NET's framework class library, whose very high-level view is illustrated in Figure 5.

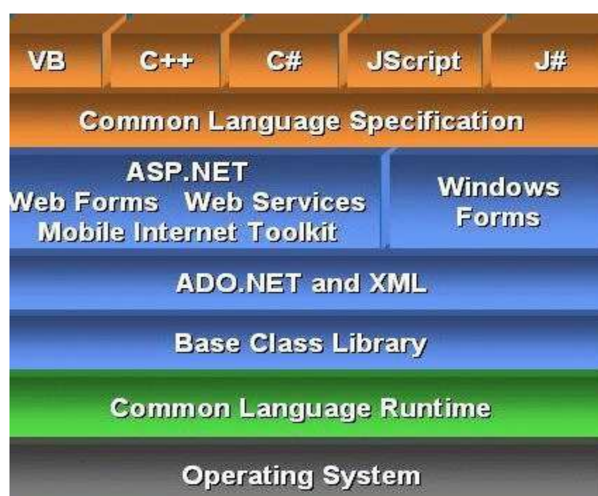

Figure 5: High-level view of .NET Framework functionality

As already described above, .NET Framework is based on CLR (Figure 5). Besides its platform independency explained above, it also offers programming language interoperability. Systems implemented on .NET framework can be programmed using any .NET language. Moreover, system parts in .NET can be implemented in different .NET languages and easily collaborate. All .NET languages must obey the Common Language Specification. This also enables a remarkable feature of cross-language inheritance. Figure 5 shows five major .NET languages: VB.NET, Managed C++, C#, JScript.NET and J#, As of now there are about forty .NET languages [46], which can be used for implementing applications on .NET framework.

Above CLR in Figure 5 the base class library is shown. It contains subsystems for I/O, communication, reflection, concurrency, serialisation etc.

Above the base class library in Figure 5 reside subsystems for database connectivity (ADO.NET) as well as XML processing.

Above these subsystems, ASP.NET functionality for web development resides. It comprises subsystems for web application as well as web services development.

Finally, Windows Forms subsystem provides APIs for constructing user interfaces for desktop applications. In addition, Windows Forms allow for rendering of simple shapes such as

ovals and polygons and enables developers to control the user-interface layout and fonts used by their applications.

To find out subsystems using which a component can incur environmental dependencies or influence its threading model, we need to explore the three layers of subsystems from Figure 5 in greater detail. The detailed .NET class library at a glance is shown in Figure 6.

Figure 6: .NET Class Library at a Glance

The correspondence of Figure 6 and Figure 5 can be seen in the following relationship:

Figure 5	**Figure 6**
Base Class Library	System namespace
ADO.NET XML	System.Data namespace System.XML namespace
ASP.NET Windows.Forms	System.Web namespace System.Windows.Forms and System.Drawing namespaces

We now get deeper into the .NET Framework class library and elaborate on subsystems from Figure 6.

At the bottom of the .NET class library in Figure 6 resides the System subsystem. It included the following namespaces of subsystems:

System.Collections The System.Collections namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hashtables and dictionaries.

System.Configuration The System.Configuration namespace provides classes and interfaces that allow programmatically accessing .NET Framework configuration settings and handle errors in configuration files (.config files).

System.Diagnostics The System.Diagnostics namespace provides classes that allow interaction with system processes, event logs, and performance counters.

System.Globalization The System.Globalization namespace contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings. These classes are useful for writing globalized (internationalized) applications.

System.IO The System.IO namespace contains types that allow reading and writing to files and data streams, and types that provide basic file and directory support.

System.Net The System.Net namespace provides a simple programming interface for many of the protocols used on networks today.

System.Reflection The System.Reflection namespace contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.

System.Resources The System.Resources namespace provides classes and interfaces that allow developers to create, store, and manage various culture-specific resources used in an application.

System.Security The System.Security namespace provides the underlying structure of the common language runtime security system.

System.ServiceProcess The System.ServiceProcess namespace provides classes that allow implementing, installing, and controlling service applications. Services are long-running executables that run without a user interface.

System.Text The System.Text namespace contains classes representing ASCII, Unicode, UTF-7, and UTF-8 character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes;

System.Threading The System.Threading namespace provides classes and interfaces that enable multithreaded programming. In addition to classes for synchronizing thread activities and access to data, this namespace includes a Thread Pool that allows using a pool of system-supplied threads.

System.Runtime.InteropServices The System.Runtime.InteropServices namespace provides a wide variety of members that support interoperability with COM and native platform services.

System.Runtime.Remoting The System.Runtime.Remoting namespace provides classes and interfaces that allow creating and configuring distributed applications. It provides a number of classes to help in using and publishing remote objects.

System.Runtime.Serialization The System.Runtime.Serialization namespace contains classes that can be used for serializing and deserializing objects. Serialization is the process of converting an object or a graph of objects into a linear sequence of bytes for either storage or transmission to another location. Deserialization is the process of taking in stored information and recreating objects from it.

Above the System namespace, subsystems for database connectivity reside (Figure 6). They include the following namespaces of subsystems:

System.Data.OleDb The System.Data.OleDb namespace is the .NET Framework Data Provider for OLE DB. The .NET Framework Data Provider for OLE DB describes a collection of classes used to access an OLE DB data source.

System.Data.Common The System.Data.Common namespace contains classes shared by the .NET Framework data providers. A .NET Framework data provider describes a collection of classes used to access a data source, such as a database.

System.Data.SqlClient The System.Data.SqlClient namespace is the .NET Framework Data Provider for SQL Server. The .NET Framework Data Provider for SQL Server describes a collection of classes used to access a SQL Server database.

System.Data.SqlTypes The System.Data.SqlTypes namespace provides classes for native data types within SQL Server. These classes provide a safer, faster alternative to other data types. Using the classes in this namespace helps prevent type conversion errors caused in situations where loss of precision could occur. Because other data types are converted to and from SqlTypes behind the scenes, explicitly creating and using objects within this namespace results in faster code as well.

Above the System namespace, also subsystems for XML processing reside (Figure 6). They include the following namespaces of subsystems:

System.Xml.XSLT The System.Xml.Xsl namespace provides support for Extensible Stylesheet Transformation (XSLT) transforms. It supports the W3C XSL Transformations (XSLT) Version 1.0 Recommendation (www.w3.org/TR/xslt).

System.Xml.XPath The System.Xml.XPath namespace contains the XPath parser and evaluation engine. It supports the W3C XML Path Language (XPath) Version 1.0 Recommendation (www.w3.org/TR/xpath).

System.Xml.Serialization The System.Xml.Serialization namespace contains classes that are used to serialize objects into XML format documents or streams.

Above the System.Data namespace, ASP.NET functionality for web applications resides (Figure 6). It includes the following namespaces of subsystems:

System.Web.Services.Desctiption The System.Web.Services.Description namespace consists of the classes that enable to publicly describe an XML Web service by using the Web Services Description Language (WSDL). Each class in this namespace corresponds to a specific element in the WSDL specification, and the class hierarchy corresponds to the XML structure of a valid WSDL document.

System.Web.Services.Discovery The System.Web.Services.Discovery namespace consists of the classes that allows XML Web service clients to locate the available XML Web services on a Web server through a process called XML Web services Discovery.

System.Web.Services.Protocols The System.Web.Services.Protocols namespace consists of the classes that define the protocols used to transmit data across the wire during the communication between XML Web service clients and XML Web services created using ASP.NET.

System.Web.UI.HtmlControls The System.Web.UI.HtmlControls namespace is a collection of classes that allow creating HTML server controls on a Web Forms page. HTML server controls run on the server and map directly to standard HTML tags supported by most browsers.

System.Web.UI.WebControls The System.Web.UI.WebControls namespace is a collection of classes that allow creating Web server controls on a Web page. Web server controls run on the server and include form controls such as buttons and text boxes. They also include special purpose controls such as a calendar.

System.Web.Caching The System.Web.Caching namespace provides classes for caching frequently used data on the server.

System.Web.Configuration The System.Web.Configuration namespace contains classes that are used to set up ASP.NET configuration.

System.Web.Security The System.Web.Security namespace contains classes that are used to implement ASP.NET security in Web server applications.

System.Web.SessionState The System.Web.SessionState namespace supplies classes and interfaces that enable storage of data specific to a single client within a Web application on the server. The session-state data is used to give the client the appearance of a persistent connection with the application. State information can be stored within local process memory or, for Web farm configurations, it can be stored out of process using either the ASP.NET State service or a SQL Server database.

Above the System.Xml namespace, subsystems for drawing reside (Figure 6). They includes the following namespaces of subsystems:

System.Drawing.Drawing2D The System.Drawing.Drawing2D namespace provide advanced 2-dimensional and vector graphics functionality.

System.Drawing.Imaging The System.Drawing.Imaging namespace provides advanced imaging functionality.

System.Drawing.Printing The System.Drawing.Printing namespace provides print-related services.

System.Drawing.Text The System.Drawing.Text namespace provides advanced typography functionality. Classes in this namespace allow users to create and use collections of fonts.

Above the System.Drawing namespace, subsystems for GUI and .NET components design reside (Figure 6). They includes the following namespaces of subsystems:

System.Windows.Forms.Design The System.Windows.Forms.Design namespace contains classes that support design-time configuration and behavior for Windows Forms components. These classes consist of: Designer classes that provide support for Windows Forms components, a set of design time services, UITypeEditor classes for configuring certain types of properties, and classes for importing ActiveX controls.

System.ComponentModel The System.ComponentModel namespace provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components.

3.4 Summary

In this chapter we have surveyed two comprehensive frameworks offering rich functionality for component development; J2EE and .NET Framework. They both allow development of desktop as well as web applications. We have provided an overview as well as some deep insight into the class libraries of the two frameworks.

In the course of the work on deployment contracts for components we indeed analysed the class libraties of J2EE and .NET in detail. We went through the libraries to identify subsystems where a component can incur environmental dependencies or influence its threading model.

Moreover, we analysed JSP or Servlet technology as well as ASP.NET technology to derive properties of the web environment components of web applications are executed in. We also used both frameworks to understand the properties of the desktop environment.

Since we based our research on the two today wide spread comprehensive frameworks, we can be confident about wide applicability of the research performed. The remainder of this work refers a lot to J2EE and .NET frameworks. We develop attributes for components based on the frameworks' class libraries in Section 4 (in particular Section 4.3). Furthermore, we define properties of desktop and web execution environments based on technologies from the frameworks in Section 5. Finally, we provide a framework for analysis of deployment contracts of components in Section 7. In the analysis we use the attributes of components from Section 4.3 as well as properties of execution environments from Section 5. Therefore, J2EE and .NET framework accompany us throughout this work.

In the next chapter we go on to define deployment contracts for components.

4 Deployment Contracts for Components

In this chapter we begin by defining dependencies a component can have. Subsequently, we move on to define a pool of attributes able to express those dependencies. Eventually, we apply the attributes to components by giving some examples.

4.1 Component Dependencies

Component definitions we give in Section 2.1.1 suggest that among other properties a software component is an independent unit of deployment not bound to a specific client and can be composed with any other component or deployed to any system. Thus, a software component is supposed to be usable in virtually any environment. However, we argue that component behaviour may be dependent on the environment it is integrated in. Therefore, we think that

component's dependencies to the environment must be made explicit to be able to integrate a component properly into a system.

Without exposing component's dependencies to the environment, static analysis about its behaviour at component deployment time (defined in Section 2.1.2) is impossible and all issues caused by a component relying on some environmental features unavailable or simultaneously used by other components can only be discovered at runtime.

Moreover, 2 components with the same interface showing operations offered by components and no information about environmental dependencies are indistinguishable at deployment time but can well behave completely different at runtime.

Information about component's dependencies to the underlying environment can be used to reasonably choose a component for a system from a pool of components and to prevent conflicts during component's composition with other components.

In the next section we undertake an attempt to identify general component dependencies. We apply them to representative component models in subsequent sections.

4.1.1 Possible General Component Dependencies

In this section we attempt to define general component dependencies a component can possess. Consider Figure 7 illustrating two components 'Component 1' and 'Component 2'.

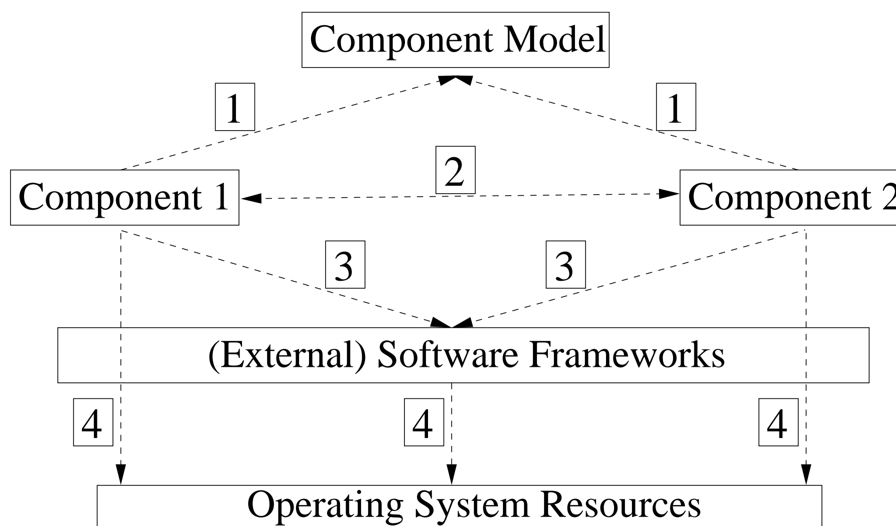

Figure 7: Possible General Component Dependencies

Component dependencies in Figure 7 are divided into 4 categories depicted by the corresponding numbers. They are:

1. Dependencies to the component model components are conform to
2. Dependencies between components themselves
3. Dependencies to (external) software frameworks
4. Dependencies to operating system resources

In the following we elaborate on each of the dependency categories defined.

Category 1: Dependencies to a component model

Component definition we adopted in this work in Section 2.1.1 suggests that a component should conform to a component model in order to qualify to be a software component of that component model. A component model imposes certain rules on its components. In other words, a component model defines a basic structure of its components. Therefore a component is influenced by the component model it is conform to and can be said to be dependent on it. This dependency, however, is not an environmental one as it is induced not by the component itself but by the respective component model. Every component of a particular component model will have the dependency as it is the prerequisite to qualify as a component of the component model.

The 2 components from Figure 7 are dependant on component model. This is illustrated by the number 1 in square on the arrows pointing from the components to component model.

Category 2: Dependencies between components themselves

A component model not only defines the basic structure of components but also composition rules among them. The composition rules used by current component models allow a component to leverage other components. Thus, a component can draw on other components to implement its functionality. In this case a component is dependent on other components.

Dependencies like this are common in today's CBSD approaches. In fact, component composition tends to introduce such dependencies as it is all about bringing components together.

Some component models follow object-oriented programming (OOP) and let composition operators for components that are classes be nearly the same what they are in OOP. In OOP, classes are allowed to make use of each other in nearly arbitrary way [19]. That is, given 3 classes A, B and C, the class A can make use of the classes B and C, the class B can make use of the classes A and C and the class C is allowed to make use of the classes A and B. If components are equated to classes, systems with no clear structure but nets of components can be built making it difficult to reason about the system. These problems have been recognised [1] and several approaches have been put forward [2, 39] to disentangle components while achieving component composition. They can be generalized as Architecture Description Languages (ADL). In ADLs there is a concept of Connectors, which can be thought of as software entities connecting components. The connectors in ADLs disentangle components in that they do not invoke each other but communicate solely via connectors. Although this approach disentangles components, it does not make them truly self-contained, independent and reusable. However, we do not elaborate on these issues in this work.

It is important to point out that this category covers dependencies between components from the same component model as shown in Figure 7. It is, however, possible that a component from one component model is using a component from another component model from within its code. Those dependencies belong to environmental dependencies of components and must be made explicit.

The 2 components from Figure 7 are dependant on each other. This is illustrated by the number 2 in square on the arrow between the components. The arrow is intentionally bidirectional. By that we show that dependencies between components could even be mutual.

Category 3: Dependencies to (external) software frameworks

Dependencies to software frameworks happen when a component utilizes some functionality offered by a framework. Examples of those frameworks include advanced graphics rendering frameworks, frameworks for accessing hardware like e.g. printers or mobile phones, frameworks for accessing data bases, web services or frameworks for mathematical calculations etc. Those frameworks can be either external or embedded in the standard framework library used for developing components. Our survey from Section 3 reveals today's most comprehensive general frameworks: J2EE and .NET. When developing components with the aid of J2EE or .NET, component developer is supported by a powerful general internal framework but they surely can draw on domain-specific external frameworks to be installed separately. J2EE and .NET are non domain-specific frameworks. Therefore, component developers may decide to additionally use external frameworks to realize required domain-specific functionality. For common software applications and components, however, one of the 2 general frameworks from Section 3 should be sufficient to develop desired functionality due to the frameworks' comprehensive nature.

Though, it is clear that all existing external frameworks cannot be investigated, components' environmental dependencies to them must be stated explicitly. It is, however, possible to look into 2 major general comprehensive frameworks, J2EE and .NET, in detail to spot the places where components can incur dependencies to the underlying environment when using them. As these frameworks are general and comprehensive, a great deal of environmental dependencies can be derived from their precise analysis. Owing to possible precise analysis of J2EE and .NET as opposed to the one of domain-specific frameworks, accurate possible dependencies to the environment can be derived. These can be used for analysing components' runtime behaviour at component deployment time enabling predictions about component's runtime behaviour. This is exactly our endeavour in this work and we expand on that below.

The 2 components from Figure 7 make use of and are dependent on software frameworks, which can be external. This is illustrated by the number 3 in square on the arrows pointing from the components to the frameworks.

Category 4: Dependencies to operating system resources

Dependencies to operating system resources originate from utilizing resources offered the component developer by the operating system. Thus, even if components are decoupled in that they are not calling each other (Category 2), they can still be coupled via the operating system when using shared operating system resources. Those subtle but vital dependencies can cause significant troubles at runtime and typically lead to system crashes or hang-ups, which are very tough to find the root cause for.

As environmental dependencies of components in this category emerge from drawing on operating system features, the operating system abstractions, which can be used by component developer, must be investigated. Different operating systems offer different abstractions. However, today components can be developed using frameworks, which embrace the operating system. Those frameworks are usually operating system independent, i.e. abstractions offered can be mapped to any operating system. This is achieved by providing a runtime environment for each operating system and a compiler which compiles the framework code into intermediate byte code, which is then translated into the native byte code by a Just-In-Time compiler of the operating system specific runtime. J2EE and .NET frameworks surveyed in Section 3 are frameworks of this kind. Therefore, by studying J2EE and .NET we are able to identify

component dependencies in this category regardless of the operating system.

The 2 components from Figure 7 make use of operating system resources through frameworks they draw on. This is illustrated by the number 4 in square on the arrows pointing from components through frameworks to the operating system resources. The frameworks can in turn make use of operating system resources. This is illustrated by the number 4 in square on the arrow pointing from frameworks to the operating system resources.

Now that, we have defined possible general component dependencies of components. We do not claim that all components must have these dependencies. We merely state that those are possible ones.

Dependencies to Component Model (Category 1) are compulsory if talking about components obeying the definition given in Section 2.1.1. We need to mention, though, that other components not following the definition, like Components-Off-The-Shelf (COTS) [6] generally do not have these dependencies.

Dependencies between components themselves (Category 2) are optional. If a component is self-contained and independent of other components, it does not have dependencies from this category.

Dependencies to (external) software frameworks are optional too. If a component makes use only of functionality offered by the programming language it is implemented in, it does not utilize any framework and is free of dependencies in this category.

Dependencies to operating system resources are also optional. If a component does not use any resource offered by the operating system (indirectly through a framework as shown in Figure 7), it is free of dependencies in this category. At this stage, we need to state that in this work we consider Random Access Memory (RAM) as a resource which is unlimited and do not regard it to be a dependency to operating system resources if a component allocates memory. In fact, modern programming languages like Java or any .NET language hide memory management from developer. Therefore, it is realistic to assume that memory is unlimited and always available to the system. It is obvious, though, that if memory cannot be allocated, the system will not be able to execute.

To illustrate the applicability of the categories of component dependencies defined in this section, in the next sections we apply them to components from Java Beans, EJB as well as .NET component model.

4.1.2 Application of General Component Dependencies to JavaBeans

In Java Beans component model [16, 58, 60, 61] components are beans, which are Java classes that emit events. Composition of the beans is done by putting event adapter classes for event dispatching in between the connected beans.

With respect to component dependencies, Java Beans can have the 4 categories of dependencies specified above. Figure 8 shows 2 Java Beans with corresponding dependencies.

The 4 Categories of general component dependencies from Section 4.1.1 can be applied to Java Beans Component Model as follows:

1. Dependencies to the component model components are conform to

   In java Beans Component Model these dependencies can be seen in the structure of components. A Java Beans must be a Java class that emits events.

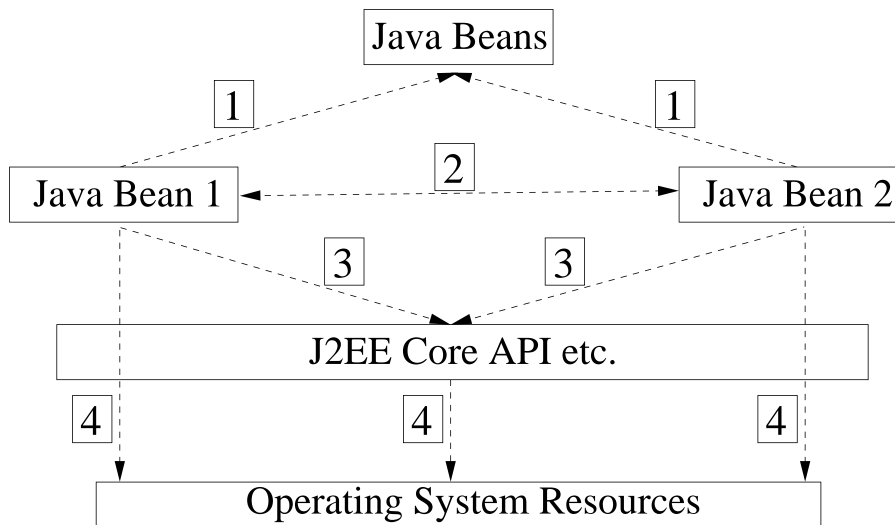

Figure 8: Component dependencies in JavaBeans

2. Dependencies between components themselves

   Java Beans components are developed independently and are in principle independent of each other. They are composed at their deployment time by event adapter classes. They are dependant on each other in the sense that they originate events to each other. In other words, although a bean is independent of another one, they can hardly be reused independently as they do not embody self-contained reusable entities. A bean needs another bean it originates events to to perform some task.

3. Dependencies to (external) software frameworks

   Java Beans Component Model is part of J2EE Framework presented in Section 3.2 and Java Beans can make use of its functionality. Additionally, they can use other external frameworks.

4. Dependencies to operating system resources

   Java Beans can utilize operating system resources by using J2EE Framework.

4.1.3 Application of General Component Dependencies to EJB

In EJB Component Model [23, 42, 13, 44, 54] components are enterprise beans, which are Java classes derived from special base classes. They are composed using method calls at their design time.

With respect to component dependencies, enterprise beans can have the 4 categories of dependencies specified above. Figure 9 shows 2 Java Beans with corresponding dependencies.

The 4 Categories of general component dependencies from Section 4.1.1 can be applied to EJB Component Model as follows:

1. Dependencies to the component model components are conform to

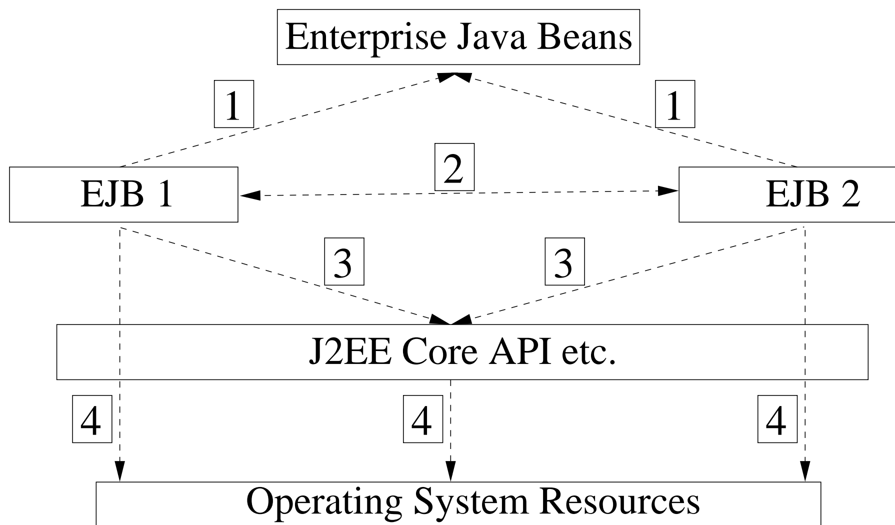

Figure 9: Component dependencies in EJB

In EJB Component Model these dependencies can be seen in the structure of components. An enterprise bean must be a Java class that derives from a special base class offered by the component model.

2. Dependencies between components themselves

   Enterprise beans are tightly coupled with each other by direct method calls. They get to know each other at their design time.

3. Dependencies to (external) software frameworks

   EJB Component Model is part of J2EE Framework presented in Section 3.2 and enterprise beans can make use of its functionality. Additionally, they can use other external frameworks.

4. Dependencies to operating system resources

   Enterprise beans can utilize operating system resources by using J2EE Framework.

4.1.4 Application of General Component Dependencies to .NET Component Model

In .NET Component Model [12, 11, 41, 52] components are classes in a .NET language implementing a special interface. They are composed using method calls. The composition can be done either like in EJB at component design time and/or like in Java Beans at component deployment time.

With respect to component dependencies, .NET components can have the 4 categories of dependencies specified above. Figure 10 shows 2 .NET components with corresponding dependencies.

The 4 Categories of general component dependencies from Section 4.1.1 can be applied to .NET Component Model as follows:

1. Dependencies to the component model components are conform to

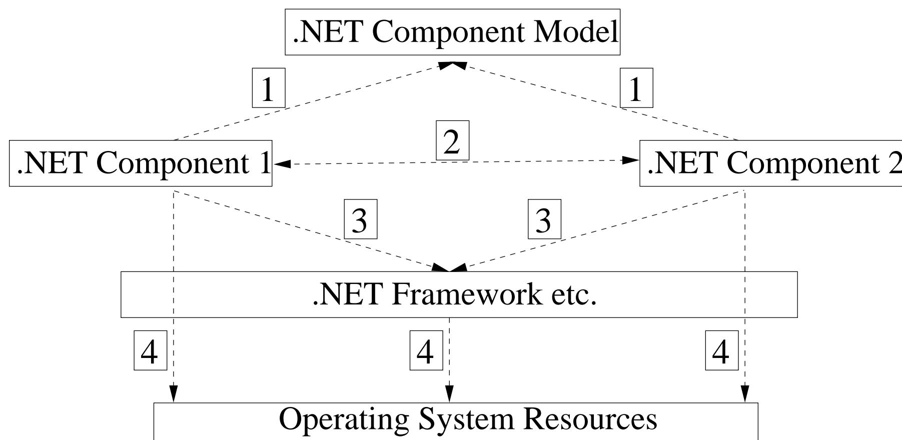

Figure 10: Component dependencies in .NET Component Model

In .NET Component Model these dependencies can be seen in the structure of components. A .NET component from .NET Component Model bean must be a class in a .NET language that derives from a special base class offered by the component model.

2. Dependencies between components themselves

   .NET components can depend on each other. They can be composed with each other by direct method calls either at their design or deployment time.

3. Dependencies to (external) software frameworks

   .NET Component Model is part of .NET Framework presented in Section 3.3 and .NET components from .NET component model can make use of its functionality. Additionally, they can use other external frameworks.

4. Dependencies to operating system resources

   .NET components from .NET component model can utilize operating system resources by using .NET Framework.

Now that, we have defined possible general component dependencies (Section 4.1.1) and showed their applicability to today's established component models Java Beans (Section 4.1.2), EJB (Section 4.1.3) and .NET component model (Section 4.1.4). We now go on to develop attributes that can reflect the defined general dependencies. As mentioned above we draw on J2EE and .NET frameworks for that purpose. Having defined a pool of attributes a component will be able to attach the relevant ones and expose its dependencies in addition to its interface.

4.2 Attributes for Components

To develop attributes reflecting component dependencies defined above we have thoroughly investigated subsystems of J2EE and .NET frameworks. We present the attributes for components derived from this analysis in square brackets reflecting their .NET implementation. Attributes are parameterisable entities. Their parameters are presented in brackets after the attribute's name. For instance:

```
[DeploymentContractAttribute(Parameter1, Parameter2)]
```

Denotes an attribute with the name 'DeploymentContractAttribute' and two parameters 'Parameter1' and 'Parameter2'. Attributes like this can be attached to components at their design time and retrieved at deployment time.

Furthermore, for attributes we define two sets 'AttributeTargets' and 'AllowMultiple':

```
AttributeTargets: {Component, Method, Property, Parameter,
ReturnValue, *}
```

and

```
AllowMultiple: {yes, no}
```

The set 'AttributeTargets' specifies the component elements on which an attribute is valid to apply. They are:

Component Denotes an attribute that can be applied to the whole component. Such an attribute denotes a property of the entire component.

Method Denotes an attribute that can be applied to a method of component. Such an attribute denotes a property of the method of component.

Property Denotes an attribute that can be applied to a property of component. Such an attribute denotes a property of a component's property.

Parameter Denotes an attribute that can be applied to a method parameter. Such an attribute denotes a property of a method parameter.

ReturnValue Denotes an attribute that can be applied to the return value of a method. Such an attribute denotes a property of the return value.

***** Denotes an attribute that can be applied to a combination of elements presented above. For instance, an attribute may be applicable either to the whole component or a component method or property.

The set 'AllowMultiple' is complementary to the set 'AttributeTargets'. It denotes whether an attribute can be applied to the specified targets multiple times.

For instance, the following attribute

```
[DeploymentContractAttribute(Parameter1, Parameter2)]
 AttributeTargets: Method, Property
 AllowMultiple: yes
```

denotes the attribute 'DeploymentContractAttribute' that can be applied to a method or a property of component. Furthermore, the attribute can be applied multiple time to the specified attribute targets.

When deriving an attribute we provide subsystem's name from J2EE and .NET (or either of them) where the environmental dependency is found. The attributes represent possible environmental dependencies a component can incur when using the corresponding subsystem. This does not mean that when using a subsystem, a component will automatically incur an environmental dependency. The attributes show environmental dependencies a component can

potentially incur. By having a pool of attributes component developer has the foundation for showing environmental dependencies of components by attaching relevant attributes to them. If an attribute is attached to a component, it means that the component definitely possesses the environmental dependency denoted by the attribute.

As the frameworks analysed are comprehensive we can be sure about the wide applicability of the derived attributes for component's environmental dependencies. In the following we present the attributes and explain them.

With respect to Categories of component dependencies defined in Section 4.1, the largest number of dependencies to be expressed by attributes comes from the usage of operating system resources (Category 4) and software frameworks (Category 3) by components. Dependencies to component model (Category 1) and to other components (Category 2) can be expressed using a few attributes.

First of all, we consider resources available to components to be of the following kinds:

`Mandatory or Optional:` A resource is mandatory required by a component if it cannot fulfil its task without that resource. On the other hand, a resource is optional if the component can do without the resource although if the resource is available it will make use of it. Using this information we can check if the component can perform in an environment where the resource in question is not available.

`Read, Write, Delete Access or combinations of them:` A resource is used for reading, writing or both. A resource can furthermore be deleted by a component. Using this information we may be able to indicate whether components using the same resource could possibly interfere with each other.

`Existence check:` Indicates whether a resource is checked for existence or not before accessing it. Using this information we may be able to prevent a component from accessing a non existent resource by putting another component creating the resource before the component requiring the resource in a component composition.

`Creation policy:` Indicates whether a resource will be created or not if it turns out to be non existent. Using this information we may prevent some resources (or components) from being created several times. This may be problematic with the usage of singletons. ???

We define the following sets reflecting the kinds of resources to be used by the attributes defined below:

```
UsageNecessity: {Mandatory, Optional}

UsageMode: {Read, Write, Delete, *}

Existence: {Checked, Unchecked}

Creation: {Yes, No}
```

In the course of defining deployment contracts we will define some other sets in addition to these ones.

In the next sections (Section 4.2.1-Section 4.2.44) we present the attributes we have developed by analysing J2EE and .NET Frameworks from Section 3.

4.2.1 Compiler input and output files

`Explanation:` It is possible to dynamically compile source code files. Source files in the file system serve as input to the compilation. Output files are compiled binaries.
`Environmental dependency:` File system. Specific files in the file system. (Category 4)
`Potential Issues:` File system not accessible. Required files in the file system not available. Several components in an application using the same output file name for compiling different input files.
`Indications:` Dynamic compilation is not going to be fast. Performance penalty possible.
`J2EE Subsystem:` There is no J2EE API for compiling files as such but it can be easily achieved by invoking java compiler directly from within a component.
`.NET Subsystem:` Microsoft.CSharp
`Attributes:`

```
[UsedCompilerInputFile(FullFileName, UsageNecessity, Existence)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
 AllowMultiple parameter specifies whether the indicated attribute can
 be specified more than once for a given program element (or target).

[UsedCompilerOutputFile(FullFileName, UsageNecessity, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A method of component performing compilation of a file applying the attributes:

```
 [UsedCompilerInputFile("C:\Input\code.cs",
 UsageNecessity.Mandatory, Existence.Checked)]
 [UsedCompilerOutputFile("C:\Output\binary.dll",
 UsageNecessity.Optional, Existence.Checked, Creation.Yes)]
 public boolean TransformFile(string fullFileName) {...}
```

4.2.2 Entries in a registry

`Explanation:` There are different registry stores for storing key/value pairs persistently. An example is Windows registry.
`Environmental dependency:` Registry used, Keys inside the registry. (Category 3)
`Potential Issues:` Registry not available. Keys in the registry not available. Several components writing the same key. One component requires a key while another one deletes it.
`J2EE Subsystem:` java.util.prefs
`.NET Subsystem:` Microsoft.Win32
`Attribute:`

```
[UsedRegistry(Name, Path, Key, Value, UsageNecessity, UsageMode,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A method of component reading a value from a registry:

```
[UsedRegistry("WindowsRegistry", "HKEY_CURRENT_USER\Identities",
"NAME", "", UsageNecessity.Mandatory, UsageMode.Read,
Existence.Checked, Creation.No)]
public string RetrieveCurrentUserName() {...}
```

4.2.3 Dynamically activated types

`Explanation:` There is means to dynamically create an instance of a type located in a binary.
`Environmental dependency:` File System or Network if the binary is located remotely. Binary identification consisting of its name, and possibly version, culture and public key token. (Culture indicates language and country settings used by the assembly. It is used when loading language and country specific resources like names of months or calendar settings. Public key token indicates the token of the public key the binary is signed with if at all). Type name in the binary to be activated. (Category 4)
`Potential Issues:` If several components activate the same type, problems with singletons or static content in the type may occur. A singleton [19] is a design pattern that can be used to prevent a component from multiple instantiation inside a process. The design pattern suggests offering a component an access method rather than using a constructor for component instantiation. Inside the component access method, the first thing to do is to check whether the component has been instantiated before. If not, an instance is created and returned. If yes, the existing instance is returned. By using this technique, a component cannot be instantiated but only accessed from outside. The instantiation is done only once inside the instantiation method and several instances of the component cannot be created in a process. Singletons (as well as static content inside components) may cause problems in systems requiring several distinct instances of components. For instance, if a bank system has N banks then it may want to have a component bank and instantiate it N times. If the bank component is a singleton, there will be no N instances of the bank component in the system but rather only one. To avoid this kind of issues, a component should have an attribute attached for further analysis indicating that it is a singleton. Furthermore, singletons may raise threading issues as well. If in the bank example introduced above, there are N instances of the bank component, each of them may be accessed by a different thread. If no N instances but one singleton is used in the bank system, the singleton will be accessed by multiple threads. This may cause multithreading issues in the singleton.

Use of static variables can lead to the same problems as with singletons. If a static variable is used inside component, multiple instances of the component will share the same variable. On the one hand, this may be desirable if e.g. a counter of all component instances is needed. On the other hand, however, this may cause problems if sharing is not the desired effect. For instance, if a component instance holds its database connection to a database in a static variable, another component instance will not be able to connect to another database since the database connection is shared.
`J2EE Subsystem:` java.lang
`.NET Subsystem:` System.Activator
`Attributes:`

```
Location: {Local, Remote}

[UsedActivatedType(FullFileName, Version, Culture, PublicKey,
TypeName, Location, UsageNecessity)]
```

```
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[Singleton(TypeName)]
 AttributeTargets: Component
 AllowMultiple: yes

MultipleInstantiation: {Allowed, Forbidden}

[StaticContentAvailable(TypeName, MultipleInstantiation)]
 AttributeTargets: Component
 AllowMultiple: yes
```

`Examples:` A method of component activating a type in a binary at runtime:

```
[UsedActivatedType("C:\Binaries\AlgebraicVectorEquationsSolver.dll", "",
"", "", "AlgebraicVectorEquationsSolver", Location.Local,
UsageNecessity.Mandatory)]
public object LoadAlgebraicVectorEquationsSolver(string typeName){...}
```

A Bank component which is a singleton:

```
[Singleton(BankComponent)]
public class BankComponent ...
```

A Bank Consortium component which contains some statics:

```
[StaticContentAvailable("DatabaseAccessor", MultipleInstantiation.Forbidden)]
public class BankConsortiumComponent ... ???
```

4.2.4 Framework Store

`Explanation:` .NET framework offers an internal volatile cache of key-data pairs.
`Environmental dependency:` Names of keys in the cache. (Category 3)
`Potential Issues:` Having several components storing different data in the cache under the same key, may lead to undesirable effects. If a component deletes an entry in the cache, which is needed by other components, they may not function properly.
`J2EE Subsystem:` Not available
`.NET Subsystem:` System
`Attribute:`

```
[UsedFrameworkStore(FrameworkName, StoreName, Key, UsageNecessity,
UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A method of component storing some data in the framework store

```
[UsedFrameworkStore(".NET", "AppDomain", "RequestID",
UsageNecessity.Mandatory, UsageMode.Write, Existence.Unchecked, Creation.Yes)]
public void StoreRequestID(string requestID) {...}
```

4.2.5 Configuration and Licensing of the execution environment

`Explanation:` .NET allows explicit configuration of itself for each executable started. .NET framework is preconfigured and its reconfiguration is not required. However, each executable can override certain settings of the default configuration by providing a configuration file with overridden settings. Moreover, application-specific settings can be contained in the configuration file. Furthermore, each executable can provide a licence without which it will not run. The licence is checked by the .NET framework automatically once licence checking is activated by the configuration.
`Environmental dependency:` File System. Configuration and licence files in the file system. (Categories 4 and 3)
`Potential Issues:` File System not available. Required files in the file system not available.
`J2EE Subsystem:` Not available as such. Available for Java Applets from java.applet subsystem.
`.NET Subsystem:` System
`Attributes:`

```
[UsedExecutionEnvironmentConfigurationFile(FrameworkName,
FullFileName, UsageNecessity, UsageMode, Existence)]
 AttributeTargets: Component
 AllowMultiple: no

[UsedExecutionEnvironmentLicenceFile(FrameworkName, FullFileName,
UsageNecessity, Existence)]
 AttributeTargets: Component
 AllowMultiple: no
```

`Example:` A component making use of some overridden .NET's configuration settings. It also provides a licence that must be available for the component to run.

```
[UsedExecutionEnvironmentConfigurationFile(".NET",
"./GraphicsViewer.config", UsageNecessity.Optional, UsageMode.Read,
Existence.Checked)]
[UsedExecutionEnvironmentLicenceFile(".NET", "./licence.dat",
UsageNecessity.Mandatory, Existence.Unchecked)]
public class GraphicsViewer ...
```

4.2.6 Environment variables

`Explanation:` Environment variables are offered by the operating system to set some settings like e.g. paths to frequently searched directories and can be used by components.
`Environmental dependency:` Names of environment variables. (Category 4)
`Potential Issues:` Several components using the same environment variable for various purposes and setting it to different values, may interfere with each other.
`J2EE Subsystem:` java.lang
`.NET Subsystem:` System
`Attribute:`

```
[UsedEnvironmentVariable(Name, UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component's method setting an environment variable

```
[UsedEnvironmentVariable("path", UsageMode.Write, Existence.Checked,
Creation.Yes)]
public boolean SetWorkingDirectory(string path) {...}
```

4.2.7 Database connections

`Explanation:` Connections to databases on database servers can be used to retrieve and persistently store data.
`Environmental dependency:` Network or file system depending on the database server location. Database names. User account on the database server. (Category 3)
`Potential Issues:` Network of file system not available. Database required not available on the database server. User account not available on the database server.

Some databases return a handle through which the user can interact with the database on successful connection. This handle sometimes requires thread affinity [14]. I.e. if the handle is accessed by multiple threads (concurrently or not), the database may fail. Therefore it is important to ensure that all components accessing the handle use it from within the same thread
`J2EE Subsystem:` JDBC (java.sql)
`.NET Subsystem:` System.Data
`Attributes:`

```
ConnectionType: {Trusted, Untrusted}

[UsedDatabaseConnection(DatabaseServerName, Location,
ConnectionType, DatabaseName, UserName, Password, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[ThreadAffineHandle]
 AttributeTargets: Property, Parameter, ReturnValue
 AllowMultiple: no
```

Indication: If a component accepts a thread affine handle, it cannot be used from multiple threads. This holds true even if a the method of component accepting the thread-affine handle is thread-safe.
`Examples:` A component's method establishing a database connection:

```
[UsedDatabaseConnection("tesco.servers.sales", Location.Remote,
ConnectionType.Trusted, "AccumulatedSalesDB", "tescodeveloper", "",
UsageNecessity.Mandatory, Existence.Uchecked, Creation.No)]
public string GetAccumulatedSalesNumber(int productID) {...}
```

A method of component returning a database handle requiring thread affine access:

```
public
 [ThreadAffineHandle]
 object
 EstablishDBConnection(...) {...}
```

A method of component accepting a database handle requiring thread affine access:

```
public RetrievePatientAge(
  [ThreadAffineHandle]
  object DBHandle,
  string PatientName) {...}
```

4.2.8 Database tables in relational databases

`Explanation:` Relational databases store data in relations or tables. They are widely used today.
`Environmental dependency:` Table names in a database. (Category 3)
`Potential Issues:` Absence of the table used.
`Indication:` A dependency to a database table can only happen if a database connection is available. Therefore, attributes from Section 4.2.7 are required in addition to the one defined here.
`J2EE Subsystem:` java.sql
`.NET Subsystem:` System.Data
`Attributes:`

```
[UsedDatabaseTable(DatabaseName, TableName[], UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A method of component accessing a table in a relational database.

```
[UsedDatabaseTable("Shapes", "Shading", UsageNecessity.Mandatory,
Existence.Uchecked, Creation.No)]
public string RetrieveShapeShading(string shapeType) {...}
```

4.2.9 Database Connectivity

`Explanation:` Database Connectivity is a driver to a database server that makes actual connections to the database. It can be considered as a framework, which may need to be separately installed on the computer initiating database connections. Often there are various database connectivities to a database type. They differ in performance and efficiency. E.g.: to connect to a relational database the following database connectivities can be used: JDBC, ODBC, OLE DB etc.
`Environmental dependency:` Database connectivity (driver) installed. (Category 3)
`Potential Issues:` Database connectivity (driver) not available.
`Indications:` Unsuitable driver might be chosen for the designed application resulting in inappropriate application's behaviour.
`J2EE Subsystem:` java.sql
`.NET Subsystem:` System.Data
`Attributes:`

```
[UsedDatabaseConnectivity(Name, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component making use of ODBC driver to access a database:

```
[UsedDatabaseConnectivity("ODBC", UsageNecessity.Mandatory)]
public class ImageViewer ...
```

4.2.10 Performance counter

`Explanation:` Performance counting is a mechanism by which application's performance data is collected. A performance counter is a resource that is identified on a machine through its name and category. Applications can use pre-existing performance counters as well as self-defined ones. Applications can e.g. measure the number of some important events occurred in the application over a time period and evaluate the measurements afterwards. For instance, it can be measured how many times a component has accessed the file system and how long an access took on average.
`Environmental dependency:` The name of the performance counter inside its category. Network if the performance counter is remote. (Category 4)
`Potential Issues:` Several components using the same performance counter name may produce undesirable results.
`J2EE Subsystem:` javax.management
`.NET Subsystem:` System.Diagnostics
`Attributes:`

```
[UsedPerformanceCounter(Name, CategoryName, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component
 AllowMultiple: yes

[UsedRemotePerformanceCounter(Hostname, Name, CategoryName,
UsageNecessity, Existence, Creation)]
 AttributeTargets: Component
 AllowMultiple: yes
```

`Example:` A component making use of a database, which uses a local performance counter to measure the time spent on database access over a period of time:

```
[UsedPerformanceCounter("DBAccess", "Statistics", UsageNecessity.Optional,
Existence.Checked, Creation.Yes)]
public class StatisticsCalculator ...
```

4.2.11 Log Files

`Explanation:` Logging is used to keep track of important events during program execution. Programs trace outputs they consider essential to a log file.
`Environmental dependency:` File system or network depending on the location of the log file. Log files in the file system. (Category 4)
`Potential Issues:` Having several components tracing into the same log file might be undesirable. Furthermore, if it is desired that several components are tracing into the same log file, they must use the same character encoding. If not the file may be unreadable since different byte representations of characters are used. For instance, a character in ASCII encoding is 7

bit whereas in Unicode 2 bytes.
`J2EE Subsystem:` java.util.logging
`.NET Subsystem:` System.Diagnostics
`Attribute:`

```
CharacterEncoding: {Default, ASCII, Unicode, UTF7, UTF8}

[UsedLogFile(FullFileName, CharacterEncoding, Location,
UsageNecessity, UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method
 AllowMultiple: yes
```

`Example:` A component making use of a log file

```
 [UsedLogFile("C:\temp\userinput.txt", CharacterEncoding.ASCII,
 Location.Local, UsageNecessity.Optional, UsageMode.Write,
 Existence.Uchecked, Creation.Yes)]
 public class UserInputProcessor ...
```

4.2.12 Event Logs

`Explanation:` Event logs are operating system owned resources which are accessible to all processes for sinking essential messages. They are used to find causes for system crashes and analyse system as well as machine-wide events. Event logs are supposed to be used with care since they are available to all processes and can easily get very large. Each process is supposed to use only one and the same event source name when sinking messages to the event log.
`Environmental dependency:` Event log names. Network if the event log is on a remote machine. Event source names. (Category 4)
`Potential Issues:` Event Logs not available. Network not available. Components using different event source names deployed into an application will give wrong impression about overall application events. Events belonging to one application are supposed to use the same event source name.
`J2EE Subsystem:` java.util.logging
`.NET Subsystem:` System.Diagnostics
`Attributes:`

```
[UsedEventLog(Name, UsageNecessity, Existence, Creation)]
 AttributeTargets: Component
 AllowMultiple: yes

[UsedRemoteEventLog(Name, HostName, PortNumber, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component
 AllowMultiple: yes

[UsedEventSource(Name, UsageNecessity, Existence, Creation)]
 AttributeTargets: Component
 AllowMultiple: no
```

`Example:` A component making use of an event log:

```
[UsedEventLog("Application", UsageNecessity.Optional, Existence.Unchecked,
Creation.No)]
[UsedEventSource("Viewer", UsageNecessity.Mandatory, Existence.Checked,
Creation.Yes)]
public class ViewingLayoutController ...
```

4.2.13 Processes

`Explanation:` A process can make use of another running process.
`Environmental dependency:` Process used. (Category 4)
`Potential Issues:` Requited program not available or not started. Several components trying to start the same process may result in failure or unnecessary overhead.
`J2EE Subsystem:` java.lang
`.NET Subsystem:` System.Diagnostics
`Attributes:`

```
[UsedProcess(Name, UsageNecessity, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[StartProcess(Name, Parameter[], WorkingDirectory,
EnvironmentVariable[], UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Examples:` A component using a spreadsheet application for showing calculated data:

```
[UsedProcess("Spreadsheet.exe", UsageNecessity.Mandatory,
Existence.Checked, Creation.No)]
public class Viewer ...
```

A component starting a process for another component but not making use of it:

```
[StartProcess("Spreadsheet.exe", "-f", ".",
null, UsageNecessity.Mandatory)]
public class CalculationHelper ...
```

4.2.14 Preferred Processor

`Explanation:` On a multiprocessor machine a process can express a wish to be executed on a particular processor.
`J2EE Subsystem:` Not available
`.NET Subsystem:` System.Diagnostics
`Attribute:`

```
[PreferredProcessor(Number)]
  AttributeTargets: Component
  AllowMultiple: no
```

`Example:` A component doing some calculations that are known to be faster on a particular processor:

```
[PreferredProcessor(2)]
public class DifferentialEquationsSolver ...
```

4.2.15 Directory Services

`Explanation:` Directory Services are light-weight databases for storing hierarchical data according to a schema. Examples of directory services are LDAP (Lightweight Directory Access Protocol) servers, Active Directory, JNDI (Java Naming Directory Interface) or WMI (Windows Management Instrumentation).
`Environmental dependency:` Directory Service. Network depending on the location of the directory service. (Category 3)
`Potential Issues:` Directory Service not available. Network not available.
`J2EE Subsystem:` javax.naming
`.NET Subsystem:` System.DirectoryServices
`Attribute:`

```
[UsedDirectoryService(Name, Location, SchemaName, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component method retrieving some organisational data from a directory service:

```
[UsedDirectoryService("MyLDAPServer", Location.Remote,
OrganisationalStructureSchema, UsageNecessity.Mandatory)]
public object GetOrganisationalData(...) {...}
```

4.2.16 Files containing graphics

??? To be revised or excluded
`Explanation:` Files containing graphics like bmp or jpeg.
`Environmental dependency:` File system or network depending on file's location. Files with graphics with specific properties. (Category 4)
`Potential Issues:` File system or network not available. Required file with specific image properties not available. What are the image properties???
`J2EE Subsystem:` java.awt, java.beans
`.NET Subsystem:` System.Drawing
`Attributes:`

```
[UsedGraphicsFile(FullFileName, ImageProperties, Location,
UsageNecessity, UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` An image processing component with a method writing the processed image to a graphics file:

```
[UsedGraphicsFile("./outputimage.bmp", ???ImageProperties,
Location.Local, UsageNecessity.Mandatory, UsageMode.Write,
Existence.Unchecked, Creation.Yes)]
public class ImageProcessor ...
```

4.2.17 Fonts

`Environmental dependency:` Fonts used. (Category 3)
`Potential Issues:` Required Fonts not available or not installed.
`J2EE Subsystem:` java.awt
`.NET Subsystem:` System.Drawing
`Attribute:`

```
[UsedFont(Name, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component rendering text using a specific font:

```
[UsedFont("Verdana", UsageNecessity.Optional)]
public class TextRenderer ...
```

4.2.18 Printers

`Environmental dependency:` Printer. (Category 3)
`Potential Issues:` Printer not available. Network not available if the printer is remote.
`J2EE Subsystem:` java.awt, javax.print
`.NET Subsystem:` System.Drawing
`Attributes:`

```
[UsedPrinter(Name, Location, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

In general, for any external device we can define the attribute:

```
[UsedDevice(Name, Type, Location, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component's method calculating the salary, which is printed on a printer:

```
[UsedPrinter("printer1", Location.Remote, UsageNecessity.Optional)]
public void CalculateSalary(...) {...}
```

4.2.19 COM components

`Explanation:` COM components are components from COM component model. A COM component is identified by a global unique identifier (GUID). COM requires the GUID to be available in the Windows registry, otherwise a COM component cannot be instantiated.
`Environmental dependency:` COM component registered with the registry. Registry. (Category 2)
`Potential Issues:` Registry not available. COM component not available or not registered with the registry.
`J2EE Subsystem:` Not available. Commercial external frameworks are available for accessing COM objects from Java, e.g. EZ JCOM [17].
`.NET Subsystem:` System.EnterpriseServices
`Attributes:`

```
[UsedCOMComponent(GUID, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

This attribute implies mandatory usage of the registry. Therefore, the attribute for indication of registry usage from Section 4.2.2 could be applied here as well. The framework for deployment contracts analysis will automatically check it too, though.

If a component accesses another component from a different component model, often a framework is needed that enables the access. E.g. J2EE does not offer connectivity to COM objects, but the external framework EZ JCOM needs to be used. For that purpose, we can define the attribute:

```
[UsedFramework(Name, UsageNecessity)]
 AttributeTargets: Component
 AllowMultiple: yes
```

For cross-component model component access, we can define the attribute:

```
[UsedComponent(ComponentModelName, UsageNecessity)]
 AttributeTargets: Component
 AllowMultiple: yes
```

Current component models include: JavaBeans, EJB, COM, CORBA, Koala, SOFA, Kobra, PECOS, Pin, Fractal, Robocop, .NET Component Model.

Furthermore, a component can indicate the component model it belongs to by attaching the attribute:

```
[ComponentModel(ComponentModelName)]
 AttributeTargets: Component
 AllowMultiple: no
```

`Examples:` A component making use of a COM component:

```
 [UsedCOMComponent("BAB23816-DF47-453f-989C-E349D9E4224B",
 UsageNecessity.Mandatory)]
 public class Viewer ...
```

A component using a framework:

```
 [UsedFramework("OpenGL", UsageNecessity.Mandatory)]
 public class Controller3D ...
```

A component indicating that its component model is ExoMeta:

```
 [ComponentModel(ExoMeta)]
 public class Sensor ...
```

4.2.20 SOAP components on web servers

`Explanation:` SOAP components are components accessed by the SOAP protocol. They reside on web servers in web server directories called virtual roots. They are like web services in that sense. However, in this section we do not deal with web services. Web services are implemented using infrastructure introduced in Section 4.2.35. Here we consider components built without the web services infrastructure. That is, these components cannot be discovered like web service using web services discovery process. Furthermore, they are not, unlike web services, described using Web services discovery language (WSDL).
`Environmental dependency:` Network, web server as well as web server directories. (Category 3)
`Potential Issues:` Network not available. Web server not available. Web server directory not available.
`J2EE Subsystem:` Not available ???
`.NET Subsystem:` System.EnterpriseServices.Internal
`Attributes:`

```
[UsedWebServer(Name, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedWebServerDirectory(WebServerName, WebServerDir, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component making use of a SOAP component on a web server:

```
[UsedWebServer("mri.co.uk", UsageNecessity.Mandatory)]
public class GetPatientName(...) {...}
```

A component creating a web server directory:

```
[UsedWebServerDirectory("mri.co.uk", "/mri/patients",
UsageNecessity.Mandatory, Existence.Checked, Creation.Yes)]
public class Deployer ...
```

4.2.21 String culture

`Explanation:` A string can represent culture-specific information. For instance, the writing system, the calendar used, formatting dates, sorting strings pertain to culture-specific information. Culture names follow the RFC 1766 standard and ISO standards. Examples of cultures are:

```
ar-EG       Arabic  - Egypt
ar-MA       Arabic  - Morocco
zh-HK       Chinese - Hong Kong SAR
zh-CN       Chinese - China
en-CA       English - Canada
en-IE       English - Ireland
```

```
de-AT       German  - Austria
de-DE       German  - Germany
```

`Potential Issues:` When manipulating or comparing strings containing culture-sensitive information, the culture of each string must be known. Strings containing basically the same information but in different cultures might not be reasonably compared. (Category 3)
`J2EE Subsystem:` java.nio.charset (does not cover as many cultures as .NET's System.Globalization)
`.NET Subsystem:` System.Globalization
`Attribute:`

```
[UsedStringCulture(Name)]
 AttributeTargets: Parameter, ReturnValue
 AllowMultiple: no
```

`Example:` A component's method returning current date in a specific culture:

```
 public
  [UsedStringCulture("de-DE")]
  string
  GetCurrentDate() {...}
```

Another component's method parsing the date returned and getting the month out of it, must declare the culture of the string it expects:

```
 public string GetMonth(
  [UsedStringCulture("de-DE")]
  string date)
  {...}
```

Obviously, the second component will be able to parse the string and get the month out of it as it expects the string to be in the German culture and it also is in the culture. Otherwise, the second component may have problems getting the month out of string.

4.2.22 File system

`Explanation:` File system is used to persistently store data in files, which are organised in directories.
`Environmental dependency:` File system. File names and directory names in the file system. (Category 4)
`Potential Issues:` File system not available. File required not available. Directory required not available. Simultaneous use of files by different components may cause problems with their consistency. Several components using the same file for different purposes may interfere with each other.
`J2EE Subsystem:` java.io
`.NET Subsystem:` System.IO
`Attributes:`

```
[UsedDirectory(FullName, Location, UsageNecessity, UsageMode,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
```

```
 AllowMultiple: yes
```

```
[UsedFile(FullName, Location, UsageNecessity, UsageMode, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component's method reading in some data from a file:

```
[UsedFile("./books.txt", Location.Local, UsageNecessity.Mandatory,
UsageMode.Read, Existence.Checked, Creation.No)]
public List GetBookChapters(string bookName) {...}
```

4.2.23 Message queues

`Explanation:` Message queues are used for intra-process communication. They allow processes to send, receive or pick messages. Messages remain in the message queue even after the process, which sent them to the queue, dies. Message queuing is usually implemented as a residential service. A process obtains a reference to the service and creates its own queue with a name using the reference.
`Environmental dependency:` Message queue. Network if the message queue is remote. (Category 3)
`Potential Issues:` Message queue or network not available. If several components use the same message queue names, they may mix their messages unless it is the explicitly desired behaviour.
`J2EE Subsystem:` Not available. J-Integra [26] framework is needed.
`.NET Subsystem:` System.Messaging
`Attribute:`

```
[UsedMessageQueue(Name, UserName, UserPwd, UsageNecessity,
UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component's method sending messages to a message queue. A message may contain a work item for another component to be executed later:

```
[UsedMessageQueue("MSMQ\Queue1", "anonymoususer", "",
UsageNecessity.Mandatory, UsageMode.Write, Existence.Checked,
Creation.Yes)]
public void SubmitWorkItem(...) {...}
```

4.2.24 Network-specific dependencies

`Explanation:` Both J2EE and .NET offer rich framework functionality to access the network. In this category we summarise possible environmental dependencies in this context.
`Environmental dependency:` Network. IP address of a remote host. Host names of a remote host. Ports used. Proxy servers used. Sockets. (Category 4)
`Potential Issues:` Network not available. IP addresses, host names or ports not available. Proxy servers not available.

J2EE Subsystem: java.net
.NET Subsystem: System.Net
Attributes:

```
[UsedHost(IPAddress, PortNumber, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedHost(Name, PortNumber, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedProxyServer(Name, PortNumber, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedSocket(IPAddress, PortNumber, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Example: A component's method making use of a socket:

```
[UsedSocket("130.88.196.76", "67", UsageNecessity.Mandatory)]
public void EstablishConnection(...) {...}
```

4.2.25 Reflected methods or properties of a type

Explanation: Reflection facility makes it possible to access a method or property of a type without getting hold of a reference to it. Basically, if a method is reflected, its name along with its parameters is passed to the reflection facility, which makes the invocation and returns the result. Furthermore, reflection allows for invocation of a method of a type in a binary. That is, it loads the binary, retrieves the type from it and makes the invocation.
Environmental dependency: File in the file system if the type has to be loaded from a binary. (Category 4) Reflected method (property) names of the type. (Category 3)
Potential Issues: File system not available. File required not available. Potential issues with singletons and static content as described in Section 4.2.3.
Indications: If the type whose methods and properties are reflected evolves over time and changes the methods and properties, its user will run into incompatibility issues since it has the names of methods and properties hard-coded in its code.
J2EE Subsystem: java.lang.reflect
.NET Subsystem: System.Reflection
Attributes:

```
[UsedReflectedMethod(TypeName, MethodName[], UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedReflectedProperty(TypeName, PropertyName[], UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

If the type has to be loaded from the binary first, the attributes describing the binary from Section 4.2.3 must be used as well. Analysis of deployment contracts will ensure that these attributes are used in conjunction.
`Example:` A component's method that makes use of reflection to reflect a type that is passed to it as 'object':

```
[UsedReflectedMethod("UserRequest", "GetID, GetStockItemNo, GetQuantity",
UsageNecessity.Mandatory)]
public void FillCache(object userRequest) {...}
```

4.2.26 Language resources

`Explanation:` Both J2EE and .NET allow strings to be specified as resources that are compiled into separate binaries and loaded by the framework on demand. Each binary contains mappings from a string name to its representation in the language the binary is for. There is a binary for each language supported. The string names are used in components and the current setting of language in the system causes the framework to load the actual string from the corresponding binary.
`Environmental dependency:` File system. Binaries in the file system containing language resources. Resource names in the binaries. (Categories 3 and 4)
`Potential Issues:` File system not available. Binaries required not available. Required resources in the binaries not available.
`J2EE Subsystem:` java.lang, java.util
`.NET Subsystem:` System.Resources
`Attribute:`

```
[UsedLanguageResources(ResourceName[], UsageNecessity, Existence)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Locale in this context is the same as culture in Section 4.2.21.
`Example:` A component's method making use of strings that are language-specific and loaded automatically by framework from a language-specific binary:

```
[UsedLanguageResources("CapitalOfAustria", UsageNecessity.Mandatory,
Existence.Checked)]
public string GetCapitalOfAustria() {...}
```

4.2.27 Remote method invocations

`Explanation:` Both J2EE and .NET frameworks provide an infrastructure for making invocations on remote objects. The infrastructure hides all the communication details and makes the remote object callable as if it were local. The infrastructure usually allows choosing communication protocol used to communicate with the remote object. Furthermore, it is possible to choose the serialisation method or format of requests and responses. For instance, there is a SOAP formatter, which serialises requests and responses in SOAP format and a Binary formatter, which serialises requests and responses in a binary format. Moreover, the infrastructure provides a mapping registry to map from logical names of remote objects to the actual objects. Thus, remote objects can be accessed by clients by knowing only its logical name. The

knowledge of remote object's actual location is unneeded. The underlying infrastructure looks the remote object's location up in the mapping registry and establishes a connection to it.
`Environmental dependency:` Network. Remote object on the network. Mapping registry on the network. (Category 4 and 3)
`Indications:` The communication speed depends on the communication protocol and serialization method used.
`J2EE Subsystem:` java.rmi
`.NET Subsystem:` System.Runtime.Remoting
`Attributes:`

```
[UsedCommunicationProtocol(Name)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedSerialisationMethod(Name)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedRemoteObject(LogicalName, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

A component usually does not know the mapping registry used to look up the location of remote objects. If it does, the following attribute can be used to indicate the environmental dependency:

```
[UsedRemoteObjectRegistry(HostName, Port)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component using remote method invocations on an object with a logical name 'StockQuote':

```
 [UsedCommunicationProtocol("TCP/IP")]
 [UsedSerialisationMethod("Binary")]
 [UsedRemoteObject("StockQuote", UsageNecessity.Mandatory)]
 public class StockViewer ...
```

4.2.28 Communication channels

`Explanation:` Remoting infrastructure introduced in Section 4.2.27 offers another high-level way of communication. It allows definition of channels disguising all the details of communication introduced before. A channel is made up of a specified communication protocol and request serialisation method. A remote object can be connected to a channel. Once connected, it can be accessed by clients using remoting infrastructure. (Dependency Category 3)
`Potential Issues:` A channel can be registered only once in a process. Several components trying to register the same channel will cause problems.
`Indications:` Channel properties like communication protocol and serialisation method determine the speed of access to the remote object.

J2EE Subsystem: Not available
.NET Subsystem: System.Runtime.Remoting.Channels, System.Runtime.Remoting.Messaging
Attributes:

```
[UsedCommunicationChannel(Name, CommunicationProtocol,
SerialisationMethod, UsageNecessity, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedCommunicationChannel(Name, CommunicationProtocol,
SerialisationMethod, HostName, PortNumber, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedCommunicationChannel(Name, CommunicationProtocol,
SerialisationMethod, IPAddress, PortNumber, UsageNecessity,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Example: A component using a communication channel to access a remote object:

```
[UsedCommunicationChannel("TCP_Bin_Channel", "TCP/IP",
"Binary", "diagramrendering.co.uk", "8181", UsageNecessity.Mandatory,
Existence.Checked, Creation.Yes)]
public class DiagramRenderer ...
```

4.2.29 Cryptography certificate files

Explanation: Cryptography certificate files store certificates used to encrypt and decrypt data. There are various algorithms for data encryption. They differ in the level of security offered and the time and effort needed to encrypt and decrypt the data.
Environmental dependency: File system. Required file in the file system created using a particular cryptography algorithm. (Category 4)
Potential Issues: File system not available. File required not available.
J2EE Subsystem: java.security, javax.crypto
.NET Subsystem: System.Security.Cryptography.X509Certificates
Attribute:

```
[UsedCertificateFile(FullFileName, CryptographyAlgorithm, Location,
UsageNecessity, UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Example: A component's method that signs other components with a cryptography certificate:

```
[UsedCertificateFile("./certificate.cer", "DSA", Location.Local
UsageNecessity.Mandatory, UsageMode.Read, Existence.Checked, Creation.No)]
public SignComponent(object component) ...
```

4.2.30 Residential Services

`Explanation:` Residential services or daemons are local processes constantly running in the background. They can be used by other processes to fulfil required tasks. They can be started by the operating system during boot up and shutdown on operating system's shutdown. But, they can also be started manually. Residential services can define users or user groups that are allowed to access the service.
`Environmental dependency:` Residential service used. (Category 4)
`Potential Issues:` Residential service not available.
`J2EE Subsystem:` Not available. An external framework Java Service Wrapper (JSW) with its subsystem `org.tanukisoftware.wrapper` can be used. In this case, the attribute [UsedFramework("JSW")] introduced in Section 4.2.19 is required.
`.NET Subsystem:` System.ServiceProcess
`Attributes:`

```
 Availability: {Checked, Unchecked}
 Launching: {Yes, No}

[UsedResidentialService(Name, UsageNecessity, Availability,
Launching, UsageMode)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedResidentialService(Name, UserName, UserPwd, UsageNecessity,
Launching, Availability, UsageMode)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component relying on a residential service that can establish a consistent download of files. Consistent download means that it can tolerate network malfunctions. If the network has not been available for a time period and has become available again, the download is resumed rather than started afresh or abandoned.

```
[UsedResidentialService("BackgroungIntelligentTransferService",
UsageNecessity.Mandatory, Availability.Checked,
Launching.Yes, UsageMode.Read|UsageMode.Write)]
public class LoadAdditionalVideoclips(...) {...}
```

4.2.31 Character encoding

`Explanation:` Character encoding denotes the number of bytes used to encode a single character. When using arrays of bytes for holding characters, their encoding is important for processing the bytes properly.
`Environmental dependency:` Encoding of the characters or the way they are stored in the byte array. (Category 3)
`Potential Issues:` Comparing and manipulating byte arrays containing characters encoded differently can yield inappropriate results.
`J2EE Subsystem:` java.nio.charset, java.nio.charset.spi
`.NET Subsystem:` System.Text
`Attribute:` We use the set 'CharacterEncoding' introduced in Section 4.2.11 here.

```
[UsedCharacterEncoding(CharacterEncoding)]
 AttributeTargets: Parameter, ReturnValue
 AllowMultiple: no
```

`Example:` A component's method returning a byte array containing characters:

```
public
  [UsedCharacterEncoding(CharacterEncoding.Unicode)]
  byte[]
  ReadCharacters(...) {...}
```

Another component accepting the byte array for further processing:

```
public void ProcessString(
  [UsedCharacterEncoding(CharacterEncoding.Unicode)]
  byte[] theString)
  {...}
```

These two components will be able to smoothly exchange the characters in the byte array as they use the same character encoding. The first component returns the characters in the byte array in the Unicode encoding. The second component expects the characters in the byte array to be in the Unicode encoding.

4.2.32 Threading Model

`Explanation:` Both J2EE and .NET frameworks allow for concurrent programming by spawning multiple threads of control in a component. A thread of control is an operating system resource that has its own execution stack and program counter. This means that multiple threads progress independently through the code. In other words, a piece of code can be concurrently executed by multiple threads in a multithreaded environment. This causes the need for synchronisation of data in that environment. Local variables are inherently local to each thread, and no synchronisation issues exist for these variables in a multithreaded environment. Global data in the program can be freely shared between threads, and thus synchronisation problems may exist with this kind of data. Component threading model basically defines whether it spawns its own threads of execution, whether it can be accessed by multiple threads and whether it launches callbacks to other components. (Dependency categories 4 and 3)

As of now, there are few publications on concurrency and components. Representative publications can be found in the area of design patterns dealing with concurrency and components: [56] and [10]. (The book [35] on concurrency gives an overview of the issues arising when using multiple threads.)

`Potential Issues:` As stated in [35], one of the necessary conditions for a deadlock to occur is the existence of a circular chain (or cycle) of components such that each component holds a resource which its successor in the cycle is waiting to acquire. With respect to components, the resources held are synchronisation primitives for synchronising multiple threads' access to some critical section in the code. Thus, components exchanging synchronisation primitives for preventing multiple threads from accessing some data, may run into a deadlock situation. This can happen if cyclic dependencies exist between components. For instance, if a component is waiting for another one to free the synchronization primitive, while the other one is in turn waiting for the first one also to free the synchronization primitive, a deadlock situation emerges.

Object-oriented software development allows components to register so-called callbacks with each other (both J2EE and .NET are object-oriented frameworks and cater for callbacks). A callback is a method of a component which has to be registered with another component. The other component calls the registered method when it decides to do so. The registering component does not know when the callback will be issued by the other component. With respect to threading issues it has been pointed out [10] that the callback can be issued in a different thread than the one which registered the callback.

Consider a common example described as an asynchronous design pattern [10, 56], where a component registers a callback with another one. After that it calls a method on the same component, which turns out to be asynchronous [27]. This means that the method spawns another thread of execution and makes the computation on that thread. The method itself returns immediately, whereas the newly created thread proceeds with the actual execution. During the execution the component decides to issue the callback to notify the other component. The callback is issued on another thread than the one which registered the callback and invoked the asynchronous method. The component receiving the callback may do something else when it is interrupted and gets the callback. As the callback is executed on a different thread, data corruption may occur when no synchronisation is done. It is possible to issue callbacks on the caller thread and not on the one that does asynchronous execution to avoid multithreading issues in the client component. This, however, requires additional effort. In any case, it is necessary to know whether callbacks are issued on the original caller thread or not when composing components using asynchronous invocations and callbacks.

Furthermore, it is necessary to know whether a component requires registering callbacks before certain asynchronous invocations can be launched. If the callback is not registered before, the component doing asynchronous operation will not be able to notify its client, which may be undesirable.

`Indications:` Thread affine handles like those introduced in Section 4.2.7 can only be accessed from a single thread. That is, even if multiple threads do not access the handle concurrently but sequentially, the thread affine handle cannot be used.

`J2EE Subsystem:` java.util.concurrent

`.NET Subsystem:` System.Threading

`Attributes:`

```
[SpawnThread(Number)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: no
```

indicates that a method spawns a thread (or threads).

```
[AsynchronousMethod]
 AttributeTargets: Method
 AllowMultiple: no
```

indicates that a method is asynchronous, i.e. it returns immediately without result and spawns a thread for execution. At the end of the execution, if the result has to be communicated to the client, a previously registered callback is issued containing the result.

```
[SynchronousMethod]
 AttributeTargets: Method
 AllowMultiple: no
```

indicates that a method is synchronous, i.e. it executes and returns its result. By default, a component's method is assumed to be synchronous unless the attribute [AsynchronousMethod] is applied. We do, however, include this attribute in the attribute pool for the sake of completeness.

```
[SynchronisationPrimitive(Name, Type)]
 AttributeTargets: Parameter, ReturnValue
 AllowMultiple: no
```

indicates that a method accepts a synchronisation primitive as a parameter or returns it.

```
[Threadsafe]
 AttributeTargets: Method, Property
 AllowMultiple: no
```

indicates that a method can be safely accessed by multiple threads due to use of synchronisation primitives.

```
[Reentrant]
 AttributeTargets: Method, Property
 AllowMultiple: no
```

indicates that a method can be safely accessed by multiple threads since it only operates on local data and thereby does not need to synchronise threads.

```
[CallbackRegistration(Name)]
 AttributeTargets: Method
 AllowMultiple: yes
```

indicates that a method is for registration of a callback.

```
ExecutingThread: {CallerThread, NonCallerThread}

[IssueCallback(Name, ExecutingThread, UsageNecessity)]
 AttributeTargets: Method, Property
 AllowMultiple: yes
```

indicates that a method issues a callback. The callback has to be registered before. Therefore, when composing components using composition operators, it has to be taken into consideration that before invoking the method issuing a callback, a method for callback registration has to be called. Moreover, the attribute allows specifying the thread which will execute the csllback. It can be either the original caller thread, i.e. the thread invoking the method issueing the callback, or other thread, e.g. a thread performing actual execution in an asynchronous invocation.
`Examples:` A component's method spawning a thread to read from 2 files simultaneously:

```
[SpawnThread(1)]
public void ReadData(string filename1, string filename2) {...}
```

A component's method indicating that it is asynchronous:

```
[AsynchronousMethod]
public void LoadData(...) {...}
```

This means that the method ‘LoadData’ returns immediately but spawns a thread to actually load data.

A component’s method indicating that it is synchronous, i.e. it returns after it has finished the actual execution. This attribute may be useful to be applied to components, some of whose methods are asynchronous, to ensure consistency in notation. Although, if a method does not bear the attribute [AsynchronousMethod], it is actually considered to be synchronous.

```
[SynchronousMethod]
public void LoadData(...) {...}
```

A component’s method returning a synchronisation primitive:

```
public
 [SynchronisationPrimitive("DBMutex", "Mutex")]
 Mutex
 GetSynchronisationForDB(...) {...}
```

and another component’s method accepting a synchronisation primitive:

```
public void RunTransaction
 [SynchronisationPrimitive("DBMutex", "Mutex")]
 (Mutex,
 ...)  {...}
```

Such constellation might be susceptible to a deadlock as explained above.

A component’s method indicating that it uses some synchronisation mechanism to protect data it is operating on from concurrent access of multiple threads and can safely be used in a multithreaded environment:

```
[Threadsafe]
public void LoadData(...) {...}
```

This means that data can be requested to be loaded by this method simultaneously by multiple threads.

A component’s method indicating that it does not operate on data that needs to be protected from concurrent access of multiple threads, does not use any thread-specific handles or other thread-specific resources and can therefore safely be used by multiple threads:

```
[Reentrant]
public void GetEllipseArea(int r1, int r2) {...}
```

The method is passed the values of radiuses of the ellipse and can compute the area following the formula:

$$S = \pi * r1 * r2 \tag{1}$$

Thus, is operates only on local data and can be safely invoked by multiple threads without need for synchronisation.

A component offering an asynchronous method ([27]), which will launch a callback of the calling component, if registered. The callback is issued not on the original caller thread:

```
[CallbackRegistration("DataLoaded")]
public RegisterCallbackForLoadDataCompletion(Delegate callback) {...}

[IssueCallback("DataLoaded", ExecutingThread.NotCallerThread,
UsageNecessity.Optional)]
[AsynchronousMethod]
public void LoadData(...) {...}
```

The client component of such component should register a callback if it wants to be notified of data load completion. It can do so, by passing the callback to the method 'RegisterCallbackForLoadDataCompletion'. It then can invoke the asynchronous method 'LoadData', which will return immediately and notify the client component of data load completion by calling the callback that was passed to the method 'RegisterCallbackForLoadDataCompletion'. Note that the callback will be issued not on the caller thread that registered the callback and invoked the method 'LoadData' but on the thread spawned inside the 'LoadData' method for loading data. This might lead to state corruption in the client component.

4.2.33 Thread-specific storage

`Explanation:` Each thread gets its own volatile storage area assigned. A thread's storage is called thread-specific storage [55] and is only accessible by the thread itself. A thread cannot access thread-specific storages of other threads. Thus, if a component puts some data into the thread-specific storage of a thread, it can only be accessed from within the same thread. Other threads do not have access to the data. The attempt to retrieve the data put into the thread-specific storage of one thread from within another thread will fail. This is one of the reasons why some components require thread affinity. That is, they require to be always accessed by the same thread to be able to use its thread-specific storage. If accessed by another thread, the component will not be able to make use of the data stored in the thread-specific storage of another thread and may fail. (Dependency Category 4)
`Potential Issues:` Absence of required data in the thread-specific storage due to thread switches. Several components in an application using the same slot name in the thread-specific storage for different purposes, may interfere with each other.
`J2EE Subsystem:` java.lang.ThreadLocal
`.NET Subsystem:` System.Threading
`Attribute:`

```
[UsedThreadSpecificStorage(SlotName, UsageNecessity, UsageMode,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

If a component using thread-specific storage requires thread-affinity, it can indicate that by applying the following attribute:

```
[RequiredThreadAffineAccess]
 AttributeTargets: Component, Method, Property
 AllowMultiple: no
```

Analysis of deployment contracts will be able to ensure that only one and same thread can access the component.
`Example:` A component using thread-specific storage to store some data about current request.

```
[RequiredThreadAffineAccess]
[ComponentModel(ExoMeta)]
public class RequestCache
{
  ...
  [UsedThreadSpecificStorage("RequestID", UsageNecessity.Mandatory,
  UsageMode.Write, Existence.Checked, Creation.Yes)]
  public void StoreCurrentRequestID(...) {...}
  ...
  [UsedThreadSpecificStorage("RequestID", UsageNecessity.Mandatory,
  UsageMode.Read, Existence.Checked, Creation.No)]
  public long GetCurrentRequestID() {...}
  ...
}
```

The component is required to add the attribute indicating that it requires thread affinity. This, however, is also checked by deployment contract analysis.

4.2.34 Web applications' dependencies

`Explanation:` Web applications run on web servers and their components may have special environmental dependencies. They include browser cookies sent along with requests to the web application, URLs referenced or used for redirection, files on the client as well as server side, and various state storages that exist in the area of web software applications. Basically, a web application is offered two different kinds of storage for storing data, which is available among requests to the web application: Application state storage and Session state storage. These storages are needed since a web application is accessed by client by using the stateless HTTP protocol that does not retain state among requests. To retain state for the whole lifetime of the application, application state storage is used. To retain state for a web browser (or user) session, session state storage is used. We elaborate on the state storages in Section 5.2 when we describe web execution environment for components.
`Environmental dependency:` Network. URLs used. Entries in state storages. (Category 3)
`Potential Issues:` Network not available. URLs used not available. Required browser cookies not available. Required entries in state storages not available. Several components using browser cookies or slots of state storages for different purposes may interfere with each other. Files in the file system on the client and server side are subject to potential issues described in Section 4.2.22 devoted to file system.
`J2EE Subsystem:` java.net, javax.servlet, javax.servlet.http, javax.servlet.jsp
`.NET Subsystem:` System.Web, System.Web.Caching, System.Web.Hosting
`Attributes:`

```
[UsedBrowserCookie(Name, UsageNecessity, UsageMode, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component makes use of browser cookies to store some data.

```
[UsedRedirectWebSite(URL)]
```

```
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component redirects clients to other web site specified as attribute's parameter. This information can be used by component client to check whether the web site is available before actually deploying the component into their application.

```
[UsedClientFile(FullFileName, UsageNecessity, UsageMode, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component of the web application makes use of the client's file system that accesses the web application. This may pose security threats.

```
[UsedServerFile(FullFileName, UsageNecessity, UsageMode, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component makes use of a file on the server side. This attribute is basically equivalent to the attribute [UsedFile(...)] presented in Section 4.2.22. However, we deliberately introduce the attribute [UsedServerFile(...)] since it is important for the web application to know on which side of the application the dependency comes into existence: on the client or server side.

```
[UsedApplicationStateStorage(Key, UsageNecessity, UsageMode,
Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component uses application state storage to store store data during its entire lifetime.

```
[UsedSessionStateStorage(Key, UsageNecessity, UsageMode, Existence,
Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component uses session state storage to store some data during a browser session.

```
[UsedErrorPage(URL)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

indicates that a component uses a web page shown when an error occurs.
`Examples:` A web application component's method making use of a cookie storing E-Mail address of the client for sending them E-Mails:

```
 [UsedBrowserCookie("EMailAddress", UsageNecessity.Required,
 UsageMode.Read, Existence.Checked, Creation.No)]
 public boolean notifyClientByEMail(...) {...}
```

A web application component's method performing redirection to another web site:

```
[UsedRedirectWebSite("http://webmaps.3d.co.uk")]
public void RedirectTo3DViewing(...) {...}
```

A web application component's method storing some information in a file on the client side:

```
[UsedClientFile("C:\Temp\3dViewing\RecentRequests.txt",
UsageNecessity.Optional, UsageMode.Write, Existence.Checked,
Creation.Yes)]
public void StoreClientRecentRequests(...) {...}
```

That means that the method will generate HTML code that will write recent requests of the client to the file "`C:\Temp\3dViewing\RecentRequests.txt`" on the client file system.

A web application component's method storing some information in a file on the server side:

```
[UsedServerFile("./<client>/RecentRequests.txt", UsageNecessity.Optional,
UsageMode.Write, Existence.Checked, Creation.Yes)]
public void StoreClientRecentRequests(...) {...}
```

That means that the method will store recent requests of a client "`<client>`" on the server side in the file "`./<client>/RecentRequests.txt`". Note that the signature of the methods in this and previous example are completely identical. They also do semantically the same action. However, they behave completely different at runtime. The former method stores the information about client's recent requests on the client side, whereas the latter does it on the server side. Client's file system is not always available to web applications and thus the former component may not be used in such environment. The latter method stores the information about client's recent requests on the server side. This has a disadvantage that for each user the component creates a folder and stores recent requests of the user in the folder ("`./<client>/RecentRequests.txt`"). That is, the more users the web application will have, the more folders will be created on the server side increasing the space needed for the web application on the web server.

A web application component's method storing the number of users visited the application in the application state storage:

```
[UsedApplicationStateStorage("UsersInTotal", UsageNecessity.Mandatory,
UsageMode.Write, Existence.Checked, Creation.Yes)]
public void addVisitingUser(...) {...}
```

A web application component's method retrieving items put into a user's shopping cart from the session state storage:

```
[UsedSessionStateStorage("ShoppingCart", UsageNecessity.Mandatory,
UsageMode.Read, Existence.Checked, Creation.No)]
public void GetShoppingCart(...) {...}
```

A web application component's method showing an error page in case of an error:

```
[UsedErrorPage("http://www.expedia.com/flights/scheduler/error.htm")]
public void EstablishWebServiceConnectionToFlightsScheduler(...) {...}
```

That means that the method will generate HTML code that shows the web page "`http://www.expedia.com/flights/scheduler/error.htm`" if the connection to the required web service cannot be established.

4.2.35 Web services-related dependencies

`Explanation:` Web services [47] are services on the network residing on web servers and accessed by the SOAP protocol. A web service describes the operations it offers using the Web Service Description Language (WSDL). Furthermore, a web service can be discovered by using a web services discovery process (UDDI - Universal Description, Discovery, and Integration). Both J2EE and .NET offer rich framework support in building web services.
`Environmental dependency:` Network. Web service and its operations. Description (.wsdl) and discovery (.disco) files of web services. (Category 3 and 4)
`Potential Issues:` Network not available. Web service required or its operations not available. File system not available. Files required not available.
`J2EE Subsystem:` JAX-RPC
`.NET Subsystem:` System.Web.Services
`Attributes:`

```
[UsedWebService(URL, UserName, Pwd, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedWebServiceOperation(WebServiceUrl, UserName, Pwd,
OperationName[], UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedWebServiceDescription(FullFileName, UsageNecessity)] - .wsdl
files
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedWebServiceDiscoveryFile(FullFileName, UsageNecessity)] - .disco
files
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component making use of a web service for stock quotes:

```
 [UsedWebService("https://partners.extranet.stockbroker.com/bluechips/",
 "", "", UsageNecessity.Mandatory)]
 public class StockViewer ...
```

A component's method invoking an operation on a web service:

```
 [UsedWebServiceOperation(
  "https://partners.extranet.stockbroker.com/bluechips/", "", ""
  {Authorise, GetDAXIndex}
  UsageNecessity.Mandatory)]
 public string GetDAXIndex() {...}
```

4.2.36 Advertisement files

`Explanation:` Web applications often display advertisement banners with changing contents. The contents is usually made up of a bunch of graphical files in the file system and corresponding URLs fed into an ad rotator control to be displayed in a loop.
`Environmental dependency:` File system. Files required in the file system. Network.
`Potential Issues:` File system not available. Files required not available. Network not available. (Category 3 and 4)
`J2EE Subsystem:` ?
`.NET Subsystem:` System.Web.UI.WebControls
`Attributes:`

```
[UsedAdvertisement(FullFileName[], URL[], UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A web application's component that displays advertisements of airlines.

```
 [UsedAdvertisement({BA.jpeg, Lufthansa.jpeg, KLM.jpeg},
                    {http://www.ba.com,
                     http://www.lufthansa.com,
                     http://www.klm.com},
                    UsageNecessity.Mandatory)]
 public class AirlinesAdvertiser ...
```

4.2.37 Cursors

`Explanation:` Cursors are graphical representations of the mouse device on computer screen. Each cursor must be installed before it can be invoked from within a component. A cursor can either be used by using its logical name or by referencing the corresponding cursor file.
`Environmental dependency:` Cursor used. File system. Cursor files in the file system. (Category 3 and 4)
`Potential Issues:` Cursor required not available or installed. File system not available. Cursor file required not available.
`J2EE Subsystem:` java.awt
`.NET Subsystem:` System.Windows.Forms
`Attributes:`

```
[UsedCursor(Name, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedCursorFile(Name, FullFileName, Location, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component that loads data and changes the cursor for the spell when the data is being loaded notifying that the application is being busy.

```
 [UsedCursor("WaitCursor", UsageNecessity.Optional)]
 [UsedCursor("Default", UsageNecessity.Optional)]
 public class DataLoader ...
```

4.2.38 Icons

`Explanation:` Icons are graphical files representing an application on a desktop, in a menu etc.
`Environmental dependency:` File system. Icon file in the file system. (Category 3 and 4)
`Potential Issues:` File system not available. Icon file required not available.
`J2EE Subsystem:` javax.swing
`.NET Subsystem:` System.Windows.Forms
`Attribute:`

```
[UsedIcon(FullFileName, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component's method changing the icon of the application on request:

```
 [UsedIcon("./changedIcon.ico", UsageNecessity.Optional)]
 [UsedIcon("./normalIcon.ico", UsageNecessity.Optional)]
 public void ChangeIcon(...) {...}
```

4.2.39 XML-related dependencies

`Explanation:` XML files are hierarchical Unicode files whose well-formedness can be checked against an XML schema [15]. Furthermore, it is possible to define transformations to automatically carry over an XML document compliant with an XML schema into another one compliant with another schema. A transformation lays down the rules, how to transfer parts of XML from one representation into another. The transformation itself is stored as an XML file and can be applied to XML files obeying the XML schema it is designed to make transformations from.
`Environmental dependency:` File system or network depending on the location of the XML, schema and transformation files. XML Files required in the file system or on the network. (Category 3 and 4)
`Potential Issues:` File or network not available. Files required not available. A component writing into the XML file while another one reading uncommitted changes. A component deleting an XML file which is required by another component.
`Indications:` Parsing large XML files is time-consuming. Different parsing techniques exist. Depending on an application's needs, a particular parsing technique may be appropriate.
`J2EE Subsystem:` org.w3c.dom, org.xml.sax
`.NET Subsystem:` System.Xml, System.Xml.Serialization, System.Xml.XPath, System.Xml.Xsl
`Attributes:`

```
[UsedXMLFile(FullFileName, SchemaFullFileName, Location,
UsageNecessity, UsageMode, Existence, Creation)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes

[UsedXmlParsingTechnique(XMLFileName, Technique)]
 AttributeTargets: Method, Property
 AllowMultiple: yes

[UsedXMLTransformation(SourceFullFileName, SourceFileLocation,
```

```
TransformationFullFileName, TransformationFileLocation,
TargetFullFileName, TargetFileLocation, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component making use of an XML file to maintain customer data:

```
[UsedXMLFile("./data/customer/customers.xml",
"./data/customer/customersschema.xml", Location.Local,
UsageNecessity.Mandatory,
UsageMode.Read | UsageMode.Write | UsageMode.Delete,
Existence.Checked, Creation.Yes)]
public class CustomerData ...
```

A component's method indicating the use of an XML file by employing Document Object Model (DOM) as a parsing technique:

```
[UsedXmlParsingTechnique("./data/customer/customers.xml", "DOM")]
[UsedXMLFile("./data/customer/customers.xml",
"./data/customer/customersschema.xml", UsageNecessity.Mandatory,
UsageMode.Read, Existence.Checked, Creation.No)]
public void ReadCustomerData(...) {...}
```

A component's method making an XML transformation:

```
[UsedXMLTransformation(
"./data/customer/customers.xml", Location.Local,
"./data/transformations/customerToBusinessPartner.xml", Location.Local,
"", null, UsageNecessity.Mandatory)]
public void CreateBusinessPartner(...) {...}
```

That is, the component transforms the XML file "`./data/customer/customers.xml`" into another another XML representation according to the transformation "`./data/transformations/customerToBusinessPartner.xml`". Note, that the component does not write the transformed XML into another XML file. This is done by another method of the component:

```
[UsedXMLFile("./data/businessPartners/businessPartners.xml",
"./data/businessPartners/businessPartnersschema.xml",
UsageNecessity.Mandatory, UsageMode.Write, Existence.Checked, Creation.Yes)]
public void StoreBusinessPartner(...) {...}
```

4.2.40 Audio and video content

`Explanation:` Java Applets allow audio and video content to be played from within the applet.
`Environmental dependency:` Network. Audio and video resources pointed to by URLs. (Category 4)
`Potential Issues:` Network not available. Resources required not available.
`J2EE Subsystem:` java.applet
`.NET Subsystem:` ?
`Attributes:`

```
[UsedAudio(URL, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

```
[UsedVideo(URL, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component playing audio clips:

```
 [UsedAudio("http://audio.co.uk/classic/beethoven1.mp3",
 UsageNecessity.Optional)]
 public class AudioRecorder ...
```

A component playing video clips:

```
 [UsedAudio("http://video.co.uk/classic/trailer/titanic.avi",
 UsageNecessity.Optional)]
 public class VideoRecorder ...
```

4.2.41 Colour profile

`Explanation:` A colour profile determines colours displayed on the computer screen. Colour profile data is based on the International Colour Consortium Specification ICC.1.
`Environmental dependency:` File system. File with the colour profile in the file system. (Category 4)
`Potential Issues:` File system not available. File required not available.
`J2EE Subsystem:` java.awt.color
`.NET Subsystem:` Not available.
`Attribute:`

```
[UsedColourProfile(FullFileName, UsageNecessity)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

`Example:` A component making use of particular colour profile data:

```
 [UsedColourProfile("../colorProfiles/colorProfile3D.dat",
 UsageNecessity.Optional)]
 public class Coloured3DRendering ...
```

4.2.42 Email-related dependencies

`Explanation:` Both J2EE and .NET frameworks offer functionality for sending E-Mails. To be able to send E-Mails a component must interact with a mail server using a communication protocol. The mail server usually requires authentication of the user requesting some service. The mail server itself is made up of namespaces of hierarchical message folders. Each namespace consists of several message folders. The component sending an E-Mail needs to specify E-Mail address used, character encoding used as well as character set (or culture from Section 4.2.21) used. Furthermore, a component may search for E-Mails in a mail folder using some search

pattern. Moreover, E-Mail attachments belong to environmental dependencies of components sending them.

`Environmental dependency:` Network. Mail server. Message folders in specific namespaces in the mail server. E-Mail addresses if hard-coded. E-Mail's character encoding as well as character set used. (Category 3 and 4)

`Potential Issues:` Network not available. Mail server not available. Account on which behalf a component accesses the mail server is not allowed to access it. E-Mail address used is not known to the mail server. File system. Files in the file system.

`J2EE Subsystem:` javax.mail, javax.activation

`.NET Subsystem:` System.Web.Mail

`Attributes:`

```
[UsedMailServer(URL, FolderNamespace[], MessageFolder[],
UsageNecessity)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[OutgoingEMailSettings(UserName, Pwd, Port, Protocol)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[IncomingEMailSettings(UserName, Pwd, Port, Protocol)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[UsedEmailAddress(EmailAddress)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[UsedEmailEncoding(EncodingName)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[UsedEmailCharset(CharsetName)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[UsedEmailSearchPattern(SearchPattern)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes

[UsedMailAttachment(FullFileName, UsageNecessity)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes
```

`Examples:` A component for sending E-Mails indicating the way it uses the mail service of a mail server:

```
 [UsedMailServer("https://webmail2.cs.man.ac.uk", "students", "",
```

```
UsageNecessity.Mandatory)]
[OutgoingEMailSettings("StudentsAdministrator", "", 25, "SMTP")]
public class OutgoingEMailManager ...
```

A component sending E-Mails with special offers to a company's clients from a specific E-Mail address in a specific format:

```
[UsedEmailAddress("special.offers@bookstore.ca")]
[UsedEmailEncoding("ASCII")]
[UsedEmailCharset("en-CA")]
public class AdsDistributor ...
```

A component's method sending an advertisement of a concert by E-Msil with an attachment:

```
[UsedMailAttachment("./Ads/ConcertAd.pdf", UsageNecessity.Optional)]
public void SendConcertAd(...) {...}
```

4.2.43 Hardware communication ports

`Explanation:` Hardware communication ports like parallel or serial ports allow sending information to devices connected to them. Communication with hardware ports can be disguised by higher level abstractions offered by a framework like it is .NET. J2EE allows, however, direct access to hardware ports.
`Environmental dependency:` Port used.
`Potential Issues:` Port not available.
`J2EE Subsystem:` javax.comm
`.NET Subsystem:` ?
`Attribute:`

```
[UsedCommunicationPort(PortId, UsageNecessity)]
  AttributeTargets: Component, Method, Property
  AllowMultiple: yes
```

`Example:` A component's method sending data directly to a parallel port:

```
[UsedCommunicationPort("LPT1", UsageNecessity.Mandatory)]
public void PrintDocument(...) {...}
```

4.2.44 Other attributes

If a method parameter or return type is 'object', a component can specify the type expected by applying the attribute

```
[UsedType(TypeName)]
 AttributeTargets: Parameter, ReturnValue
 AllowMultiple: no
```

Analysis of deployment contracts will be able to ensure type matching of a component's method returning the type and the one accepting it.
`Examples:` A component's method returning an object of type "BookManager".

```
public
 [UsedType("BookManager)]
 object
 GetBookManager(...) {...}
```

Another component's method responsible for book stock management accepts the "BookManager" as a parameter:

```
public void PrintBookStockItems(
 [UsedType("BookManager)]
 object bookManager,
 ...) {...}
```

If a protocol between component's methods needs to be defined, i.e. a sequence of a component's methods needs to be invoked in a certain order, the following attribute can be applied to the component's methods taking part in the sequence

```
[RequirePreviousMethodInvocation(Name)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Besides methods, properties can also take part in a protocol. Therefore the following attribute can be useful:

```
[RequirePreviousPropertyInvocation(Name)]
 AttributeTargets: Component, Method, Property
 AllowMultiple: yes
```

Analysis of deployment contracts will be able to ensure that the component's methods taking part in the sequence are called in the required order.
`Example:` Imagine two sensor components that can return values of some sensors. A third component is processing the values but requires the two sensor components to set the values from the sensors before the processing can take place. The following attributes would be applied in that case:

```
public class Sensor1
{ ...
  public int Gauge() {...}
  ...
}

public class Sensor2
{ ...
  public int Gauge() {...}
  ...
}

public class SensorsProcessor
{ ...
```

```
   public void SetValue1(int value) {...}
   ...
   public void SetValue2(int value) {...}
   ...
   [RequirePreviousMethodInvocation("SetValue1")]
   [RequirePreviousMethodInvocation("SetValue2")]
   public void LaunchProcessing(...) {...}
   ...
}
```

Deployment contract analysis will ensure that the composition of components “Sensor1”, “Sensor2” and “SensorsProcessor” is done in a way that the method “SensorsProcessor.SetValue1” as well as the method “SensorsProcessor.SetValue2” is invoked prior to the method “SensorsProcessor.LaunchProcessing”. That is, the output of the method “Sensor1.Gauge” should be fed as input into the method “SensorsProcessor.SetValue1”. The output of the method “Sensor2.Gauge” should be fed as input into the method “SensorsProcessor.SetValue2”. And only after that the method “SensorsProcessor.LaunchProcessing” can be reasonably invoked since it is ensured that the input data for that method has been set.

Another example of such a sequence is an asynchronous method requiring a callback to be registered before being invoked. The asynchronous method can be applied the attribute to ensure that deployment contracts analysis will demand a connection of the component where the callback registration method is invoked prior to the asynchronous method.

Now that, we have defined a pool of attributes for defining deployment contracts of component. That is, component developer has now a pool of over 100 attributes to expose its components’ environmental dependencies as well as threading model - both properties that are hidden in today’s components. In the next section we provide some examples of deployment contracts of components based on the attribute pool defined in Sections 4.2.1- 4.2.44.

4.3 Examples of Deployment Contracts

A Deployment Contract of component is a set of relevant metadata attributes from the attribute pool defined above attached to the component. In Section , for each attribute we give an example of its usage. In this section we provide additional examples of components with deployment contracts.

A component expressing an environmental dependency to a COM component (Fig. 11):

```
public class B
{
 [UsedCOMComponent("DC577003–3436–470c–8161–EA9204B11EBF",    (1)
 COMAppartmentModel.Singlethreaded,                           (2)
 UsageNecessity.Mandatory)]                                   (3)
 public void Method2(...) {...}
}
```

Figure 11: A component with an environmental dependency.

The component has a method “Method2” that has the attribute “UsedCOMComponent” attached. The attribute has three parameters: (1) shows the COM GUID used by the component; (2) says that the used COM component requires a single-threaded environment; (3) says

that the usage of the COM component is mandatory. Furthermore, implicitly the attribute says that the component requires access to a file system as well as Windows Registry since COM components have to be registered there with GUID.

A component creating a thread by (Fig. 12):

```
public class A
{
 [SpawnThread(1)]                    (1)
 public void Method1(...) {...}
}
```

Figure 12: A component with a defined threading model.

The component has a method "Method1" that has the attribute "SpawnThread" attached. The parameter (1) indicates the number of threads spawned. If this method is composed with another component's method requiring thread affinity, the composition is going to fail.

A component requiring thread affine access to all its methods (Fig. 14):

```
[RequiredThreadAffineAccess]
public class A
{
  public void DisplayData(int id) {...}
  public void DisplayProgress(...) {...}
}
```

Figure 13: A component with a defined threading model.

If this component is put into a multithreaded environment or composed with another component in such a way that is not always accessed by one and the same thread, it will fail to execute.

A component dealing with a callback (Fig. 14):

```
public class B
{
 [IssueCallback("RegisterProgressCallback",
 ExecutingThread.InternallyCreatedThread,
 UsageNecessity.Mandatory)]
 public void LoadData(int id) {...}
 [CallbackRegistration]
 public void RegisterProgressCallback(...) {...}
}
```

Figure 14: A component with a defined threading model.

The component issues a callback that has to be registered with the component's method "RegisterProgressCallback". The callback is issued on an internally created thread. Therefore, only methods that can be put into a multithreaded environment, can serve as a callback here.

Note that we do not check whether a deployment contract of component is actually correct in this work. We assume that if a component has some attributes assigned, they reflect the actual implementation inside the blackbox and no attributes have been maliciously assigned by the component developer to cause applications integrating the component to malfunction. An approach dealing with trustworthiness of components is presented in [24]. Components are heavily tested to find out their temporal behavior. However, the approach does not use metadata and deal with environmental dependencies of components and their threading model but rather with temporal properties of components in various situations.

Now that, we have defined component's properties with respect to their environmental dependencies as well as threading model. Components alone are no good since they themselves do not provide useful services unless composed into a system. Using components, systems can be constructed. Furthermore, using deployment contracts of components, components can be reasonably chosen for a system since not only interfaces of components are exposed but also their dependencies to the environment as well as threading model.

Moreover, compositional reasoning of components becomes possible by analysis of deployment contracts of chosen components. Conflicts arising from incompatible environmental dependencies or threading models of components can be prevented at component deployment time. That is, prediction of components' runtime behaviour in component compositions becomes possible.

As a next step, we want to consider system execution environments where systems assembled out of components can execute. To define them, we again draw on J2EE and .NET Frameworks described in Section 3. Defining system execution environments' properties has the advantage that components' deployment contracts can be checked against them to identify conflicts resulting from incompatible properties of target execution environment and components. Furthermore, such analysis can be done at component deployment time before runtime thus enabling predictions about component's runtime behavior in a particular execution environment.

5 System Execution Environments

By analysing J2EE and .NET Framework we deduce that there are two major kinds of applications that can be built using the frameworks: web applications and desktop applications ([32] also identifies these kinds of software applications). They also coin the two major execution environments for component-based systems: web environment and desktop environment. The two environments fundamentally differ in transient state and concurrency management, which is explored in this chapter.

5.1 Desktop Environment

Definition:
The desktop environment is an execution environment for systems deployed on a desktop. A system itself may be distributed and make use of web services. The important characteristic of the desktop environment is that systems in it are not deployed on web servers. Examples of such systems are desktop applications like MATLAB or Ghost Viewer.

Programs like UNIX commands, e.g. ls or ps, also fall into this category. Residential services like DHCP clients pertain to this category as well. They differ from applications like Ghost Viewer in that they do not contain graphical user interface (GUI).

Transient State Management:
Desktop Environment has a stateful nature. It is characterised by creating the system on system's startup and destroying it after the user has finished all the interactions with it. As a corollary, the desktop environment retains state [2] among requests to the system.

Concurrency Management:
In general, single threaded and multithreaded systems are possible in the desktop environment. If the system is single threaded, it uses the main thread to process a request. If the system is multithreaded, it spawns other threads in addition to the main thread to process a request. In the desktop environment the main thread is guaranteed to be the same for every request placed to the system.

5.2 Web Environment

Definition:
The web environment is an execution environment for systems deployed on a web server that allow user interaction using HTTP protocol. Examples of such systems are web portals like amazon or ebay as well as web services.

Transient State Management:
Web Environment has a stateless nature. It is characterised by clients interacting with a web server by means of the stateless HTTP protocol. Such mode of interaction is called Request-Response model [32]. An HTTP request does not maintain any relationship to previous requests issued to the web application. Once a request has been processed by the web application on the web server, sent back to the client and received by the client, the client is completely disconnected from the web server. The web server does not maintain any information about the client once the request has been processed. Moreover, the web server instantiates the web application each time it receives a request to it and destroys it when the request has been processed. As a corollary, the web environment is stateless as state in the system on the web server is not retained among requests to the system.

Different technologies for web application development deal with the statelessness of the web environment in different ways. We investigate Java Server Pages from J2EE and Active Server Pages from .NET framework. We do not thoroughly look into more traditional techniques for the development of web applications like various Common Gateway Interface (CGI) scripts since they follow the standard Request-Response model of the HTTP protocol and do not retain any state on the server side [64].

Java Server Pages Technology (JSP) from J2EE framework is a technology for building web applications. It is completely based on Java Servlet Technology. Therefore, we investigate it here to gain an understanding of how it works. Java Servlet Technology uses a special container running on the web server and serving the execution environment for web applications. The container hosts Servlets which are Java classes representing components of web application. The container prevents the servlets from being created and destroyed by the web server on each request processing cycle and takes over the lifecycle of the servlets. The container ensures by default that there is always one instance of the servlet to process all the requests from all concurrent users. In this case a servlet can accumulate state and use it for request processing.

[2] We mean transient state when referring to state in this work.

In addition, the servlet container allows for switching to another lifecycle mode called Single Thread Model. It allows creating a pool of system instances used for processing all requests of all users. In this case the container also guarantees that one system instance is accessed by only one thread per request. State management becomes more complicated in this case since there are several system instances, which can process requests to the system. The container does not guarantee that a user will always interact with the same system instance but only that a system instance will not be executed concurrently.

Active Server Pages (ASP.NET) from .NET framework is another technology for developing web applications. It also runs on the web server and serves execution environment for web applications. By default it follows Request-Response model of HTTP protocol and creates and destroys the system on each request. It is possible to retain state among requests using special techniques. We call them state retention and mean that if it is used then the system is not constructed and destroyed on each request processing but rather is retained in some storage and used for processing requests.

There are 2 kinds of transient state storage offered to ASP.NET applications: Application state storage and Session state storage. Both state storages are maintained by the ASP runtime on the web server. Application state storage is shared among all users of an application, whereas session state storage is dedicated to a user (browser) session. Both states storages are held in the memory of the web server and have their own peculiarities with respect to threading model. The same state storages are available to JSP applications and are maintained by the servlet container.

The stateless nature of the web environment, however, makes it difficult to maintain state when considering large and scalable web applications. The web environment is characterised by potentially very large number of concurrent users of the system. This can be achieved by simply using many web browsers pointing to the same URL. (There are other possibilities as well especially when a web service is used by different applications). If a web server has to maintain state for each concurrent user of the web application it will run out of resources (e.g. memory) with increasing number of users. A magnitude of thousands concurrent users is a quite realistic number in today's web applications and there are may be more at peak times. This certainly places a heavy burden on the web server and leads to its overload. (Note, that a web server can run several web applications in parallel) The response time of the web application increases rapidly with the growing number of users if the web server needs to maintain state for each user. The web application's availability gets worse in this case.

On the other hand, a stateless system is very scalable as the web application does not have to store each user's state. In other words, the resource consumption on the web server does not grow with the growing number of users. Therefore the web server will not suffer from running out of resources with increased number of users. However, if all users want to access the same resources the availability of the web application may still be an issue. We do not consider this kind of problems in this work as they are not due to inappropriate state management in the web application but resource management.

Even without using any state at all, a web server can hardly cope with thousands of concurrent users. To improve scalability of web applications, so-called web server farms [7] are used. A web server farm is made up of several web servers running the same web application. With a web server farm a request to an application is processed by the web server with the least load at the time of request. Note that the next request may be processed by another web server than the one which processed the first request. In this context if using state storage on one web server, the data stored may not be available on subsequent requests if it is processed by other

web servers of the server farm. I.e. in the context of a web server farm using (transient) state of any kind is not appropriate. Web applications designed for high scalability and usage on a web server farm are designed stateless and make their state persistent. It is worth pointing out that only very high scalable web applications need to be deployed to a web server farm. If an application is not expected to have thousands of concurrent users it can be hosted by one web server and make use of state storages offered.

Concurrency management:
In general single threaded and multithreaded systems are possible in the web environment. If the system is single threaded, it uses the main (or caller) thread to process a request. If the system is multithreaded, it spawns other threads in addition to the main thread to process a request.

In the web environment the main thread is not guaranteed to be the same for every request placed to the system (In case of web server farm it may even be another computer. We don't consider web server farm in this section, though). The web server spawns a thread for every request it receives. There is no guarantee in the web environment that the web server will always use the same thread for requests to a system. A thread is used by the web server to process a request to a system. Once the request has been processed, the same thread can be used to process a request to another system on the web server (A web server usually hosts several systems). If while the request to another system is being processed the web server gets another request to the first system, it will use another thread to process it. All in all, in the web environment there is no guarantee that the main thread will be the same for every request.

As we stressed in the previous section, systems maintaining state are not particularly good candidates for highly scalable web applications. However, let us consider the usage of state storages for systems maintaining state as it is by all means a practical decision for web applications not requiring high scalability.

As explained in the previous section, JSP retains a system instance among all requests of all concurrent users. The same behaviour can be achieved with ASP.NET when the application state storage is used to store the system. On the first request the system is created and stored in the application state storage and on subsequent requests, it is retrieved and used. With respect to threading model, this deployment scenario makes it possible for the system to be accessed concurrently by multiple threads. Since all users effectively place all their requests to the same system instance, system state (if any) has to be protected from concurrent modifications by multiple threads to avoid data inconsistencies. If system state is unprotected in concurrent multithreaded environment, one thread can read the state while another one can modify it resulting in state corruption.

Another deployment scenario with JSP is to have a pool of system instances and use one of them to process a request from a user. A system instance is guaranteed to be accessed by exactly one thread during request processing. In other words the servlet container prevents a system instance to be accessed concurrently. Thus a system instance may not protect its state from concurrent access by multithreaded threads. However, the maintenance of system state becomes difficult in this case as there is no single system instance for request processing. A request from a user may be processed by one system instance, while the next one may be processed by another one. Thus, the state accumulated during one request may not be available on the next one since they may be processed by different system instances.

Moreover, the servlet container does not guarantee thread affinity to system instances from the pool. In other words, the container may issue different requests to the system in different

threads, not concurrently though. This means that if the system contains or interacts with some components requiring thread affinity, it cannot be deployed in such environment and certainly not in a multithreaded one. It is worth noting that ASP.NET does not offer such deployment scenario per se. It might be modeled using Application as well as Session state storage but is not really common.

Another deployment with ASP.NET is to use Session state storage to store a system instance. Session state is specific to a user (browser) session. If a system instance is created on session creation, stored in the session state storage and retrieved when the user makes requests to the system during the session, a user always interacts with the same instance of the system. Different users get different system instances, though. In such deployment scenario system state need not be protected from concurrent access from different threads as a user can only access the system sequentially. However, thread affinity is not guaranteed in this case. One request may be issued in one thread, whereas another one in another thread. Therefore, components requiring thread affinity are not suitable for such deployment scenario. Note that in this case we are not dealing with concurrent access of the system by multiple threads but rather with different threads operating sequentially on the same system instance. Note that JSP.NET does not offer such deployment scenario where a system instance is created for each user session.

In this context, it is important to recognise that components making use of static variables, properties, methods or singletons may be problematic as they are shared among all system instances in the web server process. When using static entities or singletons several users may operate on the same data and affect each other's system state, which may be undesirable. The access may be concurrent depending on deployment scenario.

Finally, ASP.NET allows Request-Response deployment scenario. Here a system instance is created on each request and destroyed on request processing completion. The same model is used by various CGI scripts. They, unlike JSP or ASP.NET, do not offer frameworks which allow server-side state retention. Since a new system instance is created on each request, there are no threading issues in such deployment scenario. However, there is no state retention and systems maintaining state cannot be deployed in this way. Stateless systems, on the other hand, are suitable for such deployment, very scalable and can withstand concurrent access of large number of users.

To sum it up, the web environment imposes threading issues on single threaded systems. As shown above even if a system does not spawn its own threads of control, the web environment makes it sometimes possible for several threads to operate in the single system instance. This in turn requires state protection in the system.

To summarise system instantiation modes in the web environment found by analysing J2EE and .NET, we present them using Table 2.

Table 2 shows that system instantiation modes in the web environment range from creating a system instance per request to the system (default HTTP behaviour; Column 1) to having a system instance for all concurrent requests (like in desktop environment; Column 3). In between there is a mode where a system instance is created per user (browser) session as well as a mode where a pool of synchronised instances serve all concurrent requests to the system. All of these system instantiation modes have various impacts on system transient state as well as create different threading environment for the system to run in. In the following we concisely elaborate on each of these modes.

Column 1 of Table 2 presents a mode where the system is instantiated on each request and destroyed after it has been processed. This mode is represented by default behaviour of ASP.NET and allows for highly scalable web applications able to withstand web server farm

	1	2	3	4
	System instance per request	**System instance per user (browser) session**	**System instance for all concurrent requests**	**Pool of synchronised system instances for all concurrent requests**
ASP.NET	ASP.NET default	ASP.NET using session state storage	ASP.NET using application state storage	
JSP			JSP default	JSP using Single Thread Model
System transient state management	System state is not retained among requests	System state is retained during a user session	System state is retained among all concurrent requests	System state is not retained among concurrent requests
Concurrency management	A system instance is accessed by one thread	A system instance can be accessed by multiple threads sequentially	Concurrent access of a system instance by multiple threads	A system instance can be accessed by multiple threads sequentially

Table 2: System instantiation modes in the web environment and their effects

deployment. System state is not retained among requests in such execution environment. Since each system request is processed by a fresh system instance, it is only accessed by one thread.

Column 2 of Table 2 presents a mode where the system is instantiated once per user (or browser) session. All requests within a user session are processed by the same system instance. This mode is represented by ASP.NET when session state storage is used to store the system. System state is retained for a user session in such execution environment. A system instance can be accessed by multiple thread, although not concurrently but sequentially. In other words, no thread affinity is ensured in this execution environment.

Column 3 of Table 2 presents a mode where the system is instantiated once for all concurrent requests. This mode is represented by ASP.NET when application state is used to store the system as well as JSP's default behaviour. This mode is somewhat similar to the desktop environment with respect to state and concurrency management. System state is retained among all concurrent requests to the system. A system instance can be accessed by multiple threads concurrently.

Column 4 of Table 2 presents a mode where there is a pool of system instances serving all concurrent requests to the system. This mode is represented by JSP's Single Thread Model. System state is not retained among requests as they can be processed by different system instances. A system instance cannot be concurrently accessed by multiple threads. However, no thread affinity is guaranteed and a system instance cannot be accessed by different threads sequentially.

Now that, we have defined two execution environments for systems to run in as well as their properties considered in this work. Since we based our research on comprehensive and established frameworks, J2EE and .NET, we can be confident about wide applicability of defined

execution environments and their properties.

In the next section we want to define which resources can be available to a system's components regardless of the execution environment they run in. This equips us with more properties that can be checked against deployment contracts of components for conflict prediction.

5.3 Resource Availability

Regardless of the environment a system runs in, it may or may not have access to specific resources available on or outside the computer it is executing on. The resources can be divided into the following categories:

Network resources
Resources in the local File System
Input/Output Devices

Furthermore we can add the categories:

Local Residential Services
Local Databases
Local Event Logs
Local Message Queues
Local Performance Counters
Local Directory Services
Local Registry Storage

We consider Random Access Memory (RAM) as a resource which is unlimited and therefore do not put it into the categories of resources a system can have access to. We see the memory as a resource always available to the system. In fact, modern programming languages like Java or any .NET language hide memory management completely from systems. A system implemented in those languages cannot influence memory allocation and reclaiming as it is managed by the runtime like JVM or .NET CLR. Therefore, it is realistic to assume that memory is available to the system whenever needed. It is obvious, though, that if memory cannot be allocated to a system, for whatever reason, the system will not be able to execute.

In the following we explain the defined categories in detail.

`Network` - Network resources are either available or not in the target execution environment. This means that unless network is accessible in the target execution environment, components in a system that make use of resource not on the local machine but on the network like e.g. web services or communicate with other remote entities, cannot be used and have to be replaced by the ones without the need to access the network.

`File System` - File system is either available or not in the target execution environment. Especially web applications do not always have access to the file system on the client side. If file system is not available, components requiring it cannot be deployed in that environment.

`Input/Output Devices` - Input/Output Devices are local devices on the computer a system is deployed to, which are either available or not in the target execution environment. By this we mean all external devices a particular computer can have. Of course, this is dependant on hardware configuration and device types connected to the computer. Therefore, we do not differentiate this category in particular device types. They, however, include: Printers, USB devices like memory pens, infrared devices etc. If an input/output device is not available in the target execution environment of a system, a component requiring it cannot be used.

`Local Residential Services` - Local Residential Services are services that are continuously running in the background. Those include DHCP client servers, global message queues, security services etc. They may be available or not in the target execution environment. If a component requires a local residential service but it is not available in the target environment, it cannot be used. As we cannot survey all residential services, we only provide a flat distinction here, that is, either local residential services are available or not. If a system is deployed to an execution environment, where a residential service is not available and a component in the system, requires it, the component cannot be used.

`Local Databases` - Local Databases are databases that are installed locally on the computer the system is running on. We do not include Directory Services like e.g. Active Directory or LDAP Directories as well as registry storages like Windows Registry in this category. A component in a system requiring access to a database cannot be used in an execution environment without the database.

`Local Event Logs` - Local event logs are machine-wide logs held by the operating system for processes that want to persistently log essential messages during their execution. Event logs are widely used as storage for important events during program execution. They are used to find causes for system crashes and analyse system as well as machine-wide events at particular point in time. A component requiring access to an event log, cannot be used in a system whose target execution environment does not offer access to event logs.

`Local Message Queues` - Message Queues are used for intra-process communication. They allow processes to send, receive or pick messages. A component requiring access to a message queue, cannot be used in a system whose target execution environment does not offer access to message queues.

`Local Performance Counters` - A performance counter is a resource offered by the operating system that allows applications to accumulate performance data about themselves. A component requiring access to a performance counter, cannot be used in a system whose target execution environment does not offer access to performance counters.

`Local Directory Services` - Directory Services are light-weight databases for storing hierarchical data according to a schema. A component requiring access to a directory service, cannot be used in a system whose target execution environment does not offer access to directory services.

`Local Registry Storage` - There are different registry stores for storing key/value pairs persistently. An example is Windows registry. A component requiring access to a registry storage, cannot be used in a system whose target execution environment does not offer access to registry storage.

All the resources in categories above may or may not be available to the system in its target execution environment.

Now that, we not only have defined execution environments for systems but also resources that may be available to components considered in this work. In the following section, we summarise research done in Section 5 and point out how we can use it in conjunction with deployment contracts of components from Section 4.

5.4 Summary

When assembling a system out of pre-existent components, it is essential to find a match between the features offered by the execution environment and those required by the components in the assembly. I.e. it is important to know which execution environment the system is deployed to

and which resources are available to the system in the target execution environment.

To sum it up, we distinguish between two environments: web environment and desktop environment. The properties of them considered in this work are transient state and concurrency management in general. Table 3 shows detailed properties we use to differentiate between them.

	Desktop Environment	**Web Environment**
System instantiation	Once for all requests	Differs between technologies
Stateless	No	Yes
Stateful	Yes	No
State retention issues	No	Yes
Main thread affinity	Yes	No
Synchronisation on concurrency	May be required	May be required
Imposing multiple threads	No	Yes

Table 3: Properties of the desktop and web execution environments

As shown in Table 3, system instantiation is done once for all requests a user can place to the system in the desktop environment, whereas it differs among technologies and options within a technology in the web environment.

The desktop environment is stateful, whereas the web environment is inherently stateless by default through using stateless HTTP protocol. As a corollary of this, the desktop environment does not pose any state retention issues, whereas the web environment does. As we have seen above, special techniques like containers and state retention storages are used to retain application state among requests in the web execution environment.

With respect to concurrency management, the desktop environment ensures main thread affinity for all requests to the system, whereas the web environment does not. Both execution environments may require synchronisation when concurrency is there. Moreover, the desktop environment does not impose multiple threads on a system instance, whereas in the web environment this may take place. Although a system is single threaded, in the web environment it can be exposed to multiple threads if user requests are processed by the same system instance but different threads.

With respect to resources we distinguish the following categories that may or may not be available to components of an application in an execution environment (Table 4):

Now that, we have defined deployment contracts for components and execution environments for systems composed out of the components. This enables us to (i) reasonably choose components from component pool for a system since they expose far more information about implementation than interfaces currently do, (ii) do compositional reasoning of components by predicting conflicts on component deployment by comparing components' deployment contracts, (iii) predict conflicts of components in a system with the target execution environment. In fact, we have defined resources available or not in the target execution environments in Section 5. A particular execution environment can define a set of resources it offers to systems deployed into the execution environment. Components in our approach are augmented with deployment contracts defined in Section 4 showing their environmental dependencies or requirements on the execution environment. Thus, we can check whether a component in a system deployed to an execution environment with defined available resource can fulfil its tasks. This is done by matching deployment contracts of components expressing their environmental dependencies

Resource
Network
File System
Input/Output Devices
Local Residential Services
Local Databases
Local Event Logs
Local Message Queues
Local Performance Counters
Local Directory Services
Local Registry Storage

Table 4: Resources available to components in an execution environment

(and thus their requirements on the execution environment) and resources available in the execution environment. Note, that this is done at component deployment time predicting runtime conflicts.

What remains unchecked (iv) is system-specific properties (not component-specific ones) and properties of the execution environment the system is deployed to. We have defined desktop and web environments' properties in terms of transient state and concurrency management. In order to check whether these properties of the execution environment are suitable for a system, we need to know about the management of these properties in the system. For instance, we need to know whether the system accumulates state to check it against state management in the target execution environment.

Therefore, as a next step, we want to consider system-specific properties of systems constructed out of components. If components are composed, some emergent properties arise that are proper for the chosen composition. In the next section, we define these properties considered in this work.

6 System-specific Properties

System-specific properties are properties that emerge from assembling components into a system. When components are not assembled into the system, these properties do not exist. We have not considered the ways systems can be constructed out of components in this work. This is because it is not our primary concern here. However, it can generally be said that current research approaches to CBSD involve connectors [39, 3, 2, 21, 20, 5, 31] for components. It is important to note that components alone are not deployed to execution environments. Components are deployed or integrated into systems where they can provide useful services by collaborating with other components. To collaborate, components need to be connected by connectors, which ultimately results in complete systems that are deployed into execution environments.

When composing components into systems, some properties emerge. We consider system's transient state and threading model to be system-specific properties in this work. We do not consider other system-specific properties here. In the following chapters we elaborate on system's transient state and threading model.

6.1 System State

In CBSD, system's state can emerge when components are assembled together and run. Then components may accumulate some state based on requests placed to the system and use this state for further request processing. If no component in the system accumulates state, the system is referred to as stateless [28]. Such a system is an ideal candidate for the web environment as it creates the system afresh on each request (by default) catering for high scalability. A stateless system cannot lose any state between requests and is therefore indifferent to system creation and destruction between requests.

A stateful system, on the other hand, is less suitable for the web environment. If the system has state and it is destroyed after a request is processed, the next request will probably not be processed properly. In other words, to accommodate a stateful system in the web environment requires additional effort. It is possible to retain state between requests in the web environment, though. However, retaining state in the web environment may impair system's performance, availability and scalability.

A stateful system is suitable for the desktop environment. The desktop environment constructs the system on its start and destroys it when the user has finished all the interactions with the system. Therefore retention of state between requests is not an issue in the desktop environment.

6.2 System Threading Model

System's threading model tells us whether the system always uses one thread of control or spans multiple threads to process its requests. A decision to introduce multiple threads of control can be made to parallelise processing of requests and thus allow more requests to be processed for a time period. With introduction of multiple threads executing concurrently in a system, the need of synchronisation of access to certain places in the system by multiple threads emerges.

Stateless systems are easier to construct and maintain when multiple threads of execution are available. System's state needs to be guarded from concurrent access of multiple threads. When a thread reads system's state, it must be prevented from writing by another one at the same time to avoid data inconsistency (or state corruption) in the system. As a corollary, if the system does not accumulate state, it needs less synchronisation and is easier to implement.

In the desktop environment multiple threads can operate in a system instance only if it spawns them in addition to the main thread. The desktop environment does not introduce new threads in the system. As we have seen in the web environment this does not hold true. Being deployed on a web server implies being exposed to multiple users and potentially concurrent threads. In this case the need for synchronisation is not imposed by the system itself but by the environment. However, if the system itself spawns several threads, e.g. from within a component connector, it is a system-specific property that multiple threads operate in the system and may require synchronisation.

To sum it up, we can distinguish between stateful and stateless systems as well as multi-threaded and single threaded systems.

6.3 Summary

Table 5 summarises system-specific properties considered in this work.

We distinguish between four types of systems:

System	Stateful	Stateless
Multithreaded	A	B
Singlethreaded	C	D

Table 5: Considered System-specific properties

A - A system is stateful and multithreaded. In this system the state needs to be protected from concurrent access of multiple threads.

B - A system is stateless and multithreaded. In this system no synchronisation is required as there is no global system state.

C - A system is stateful and singlethreaded. This system does not need any synchronisation if deployed in the desktop environment. It may need synchronisation if deployed in the web environment depending on the deployment scenario there.

D - A system is stateless and singlethreaded. In this system no synchronisation is required as there is no global system state.

Now that, we have defined system-specific properties. They can be used to check whether a system can be reasonably deployed to an execution environment. In Section 5 we have defined execution environments and their properties. With the system-specific properties defined in Section 6, we are enabled to check whether a system can be smoothly deployed into the chosen execution environment. Furthermore, we can analyse properties compatibility of individual components and target execution environment.

In the next chapter, we pull research results from Section 4 on deployment contracts for components, Section 5 on system execution environments and Section 6 on system-specific properties to provide a comprehensive framework for deployment contracts analysis.

7 Framework for Deployment Contracts Analysis

Component-based software development is a promising software development paradigm for developing software out of ready-to-use building blocks - software components. When building component-based software systems the following composition of properties (or strengths) needs to be resolved: Execution environment-specific properties vs. System-specific properties vs. Component-specific properties.

Execution environment-specific properties refer to inherent properties of the target environment the system will be deployed to. The target environment may be desktop or web environment both with their own peculiarities. Furthermore, some resources may be or may be not available in the system's target execution environment.

System-specific properties refer to properties that emerge from a specific composition of components into a system. Those properties don't exist if components are unassembled. Moreover, they may not emerge even if components are assembled in another way. An example of these properties is state in the system. If components are assembled into a system, some components may contain state. If assembled otherwise state may be held by other components. System's state needs to be managed according to the environment the system is supposed to run in.

Component-specific properties refer to properties which are inherent to components themselves taking part in a system. A component can have some requirements on its execution environment. E.g. a component may inherently need to access the network. Another example is a component designed to only be used in a single threaded environment. If the component is put into a system with possible multiple threads of execution, it will not be able to run properly.

Figure 15 summarises the properties or strengths that need to be resolved in a component-based system development.

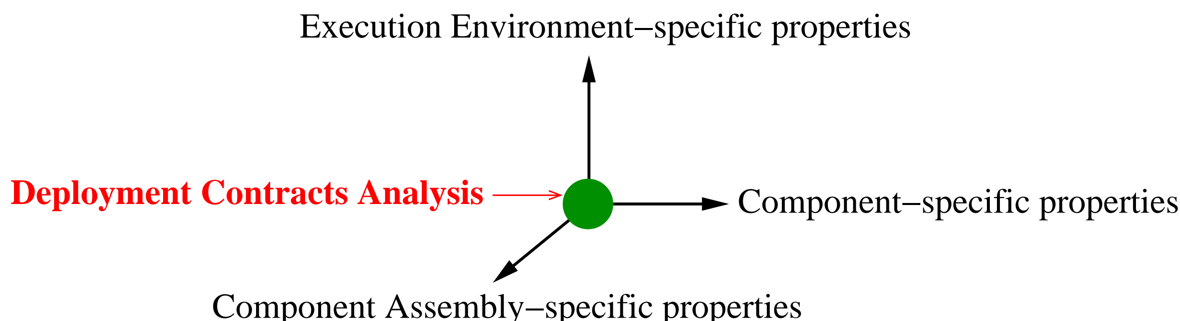

Figure 15: Properties (or strengths) to be resolved in a component-based system development

When starting a component-based system development, components for the system need to be found, chosen or developed. Each component has its own properties, which is denoted by the innermost rectangle in Figure 15. Furthermore, components need to be composed together using some composition operators, which ultimately results in a system with some defined properties. This is denoted by the middle rectangle in Figure 15. Once the system is ready, it is deployed to an execution environment with some properties. This is denoted by the outermost rectangle in Figure 15. Properties of individual components as well as system-specific properties need to be matched with those of the target execution environment.

When assembling a system out of pre-existent components, it is essential to find a match between environment-specific, system-specific and component-specific properties. It is important to do so before system runtime to be able to predict and prevent conflicts, which may arise through incompatibility of the properties in the three categories.

In previous chapters we presented research that defines component-specific, system-specific and environment-specific properties. Table 6 summarises what has been done so far and relates it to the general framework of properties to be considered in a component-based development from Figure 15.

As shown in Table 6 we have defined Deployment Contracts for software components that correspond to component-specific properties. Furthermore, we have defined system-specific properties and can distinguish between stateful and statless as well as multithreaded and singlethreaded systems and combinations of them. Finally, we have defined system's execution environments to be web and desktop environments. Properties of these environments correspond to execution environment-specific properties. In other words, we can apply the general framework of properties to be considered in a component-based system development from Figure 15 to our approach.

Our approach to component-based software development is shown in Figure 16.

Components in our approach are augmented with deployment contracts. A deployment contract of component is, like its interface, visible outside component. Furthermore, it can

	Component-specific properties	System-specific properties	Execution environment-specific properties
Deployment contracts for components (Section 4)	Yes	–	–
Stateful/Stateless and Multithreaded/Singlethreaded Systems (Section 6)	–	Yes	–
Properties of web and desktop execution environment (Section 5)	–	–	Yes

Table 6: Defined properties for a component-based system development

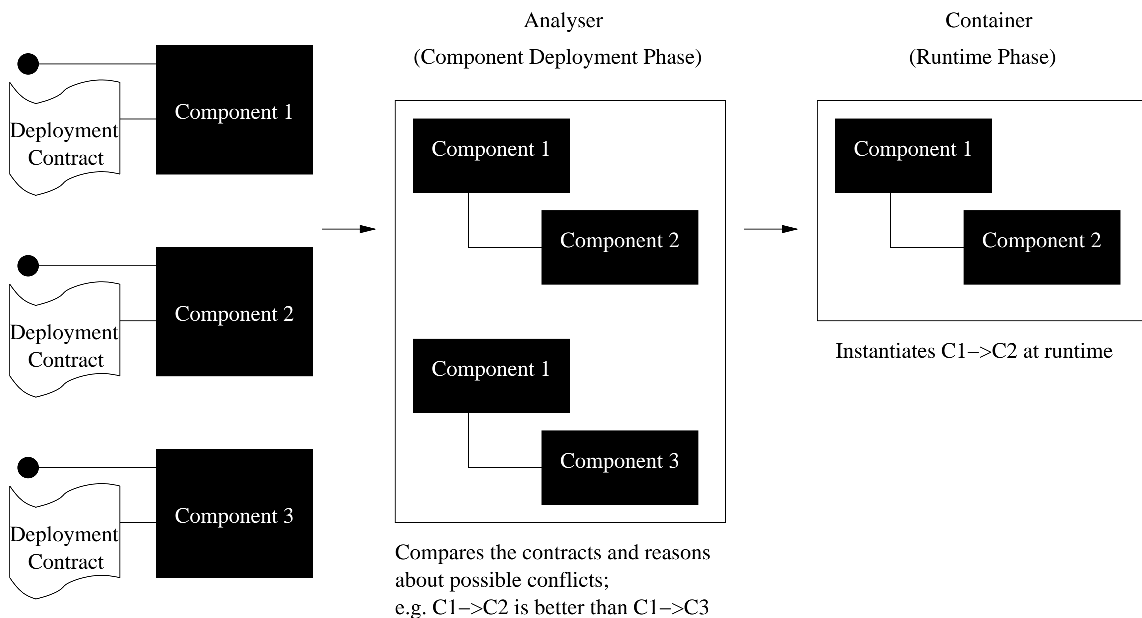

Figure 16: Overview of the proposed component model

be retrieved for analysis at component deployment time. Figure 16 shows three components: "Component1", "Component2" and "Component3" exposing their interfaces as well as deployment contracts.

Furthermore, when composing components into systems, an analyser of deployment contracts is used to:

- enable system developer to analyse deployment contracts of components when choosing them for a system. The analyser can show deployment contract of a component to system developer.

- do static compositional reasoning of components by predicting conflicts by comparing components' deployment contracts automatically and spotting and showing incompatible properties.

- enable system developer to define the target execution environment of the system based on predefined sets of properties for desktop and web environment.

- predict conflicts of components with the target execution environment. This is done by automatically comparing components' deployment contracts and properties of the target execution environment. Note that properties of the target execution environment can be flexibly changed to simulate the environment and find out in which environments a component can run.

- define system-specific properties by choosing between stateful/stateless system, multi-threaded/singlethreaded system or combinations of them according to system-specific properties defined in Section 6.3, Table 5.

- automatically check compatibility of system-specific properties and those of the target execution environment.

Finally, after the analyser has predicted and prevented conflicts the assembled conflict-free system is actually constructed by using a generic container container [30].

7.1 Example of Deployment Contract Analysis

To illustrate the usefulness of deployment contracts we show how they can be applied to a design pattern for components described in [56, 10]. The design pattern is for systems including one component that loads data in the background and another one (or more) that displays the data loaded. Furthermore, while the data is being loaded in the background, the loading component notifies the one displaying the data about the chunks of data already loaded. The component displaying data can either display the chunks of data already loaded, thus implementing so-called streaming, or just display a visualisation of it, e.g. a progress bar, which advances each time the loading component sends a notification that a chunk of data has been loaded.

Fig. 17 shows two such components. Component A has two methods "DisplayData", which

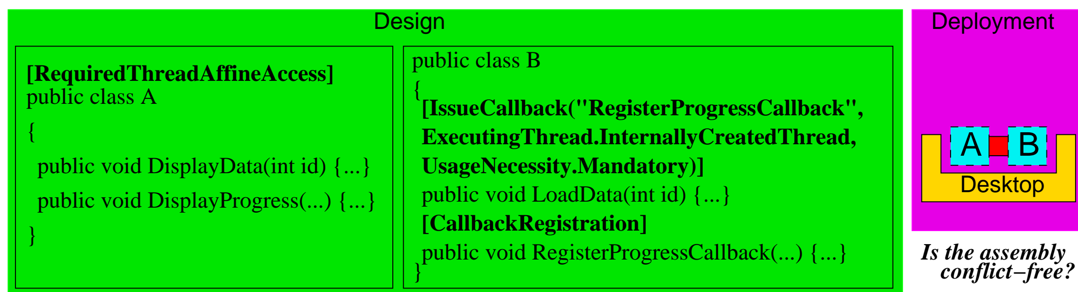

Figure 17: Implementation of a design pattern for components with use of metadata attributes

displays loaded data, and "DisplayProgress", which displays a progress bar. A's developer knows that the method "DisplayProgress" may be used as a callback method by another component, which loads the data. They also know that a callback may be invoked on different threads. Since no synchronisation of multiple threads is done inside the component, state corruption will arise if it is used concurrently from multiple threads. Therefore, in the design phase, the component developer is obliged to attach the attribute "RequiredThreadAffineAccess" at component level

(in the design phase) to let the system developer know that the component must not be used in multithreaded scenarios.

Component B has two methods: “RegisterProgressCallback” and “LoadData”. The method “RegisterProgressCallback” registers a callback of another component with the component. In this situation, the component developer is obliged to attach the attribute “CallbackRegistration” to the component’s method. The method “LoadData” loads the data. Moreover, while the data is being loaded, the method invokes a callback to notify the component’s user that a certain chunk of data has been loaded. In this situation, the component developer is obliged to attach and parameterise the attribute “IssueCallback”. The attribute parameters show that the method will issue the callback registered with the method “RegisterProgressCallback”. The thread executing the callback will be an internally created one. Furthermore, the callback is mandatory. Therefore, the component must be composed with another component in such a way that the method “RegisterProgressCallback” is called before the method “LoadData” is called.

In the deployment phase, suppose the system developer chooses the desktop as the execution environment. Furthermore, suppose the system developer decides to compose components A and B in the following way: since A displays the data and needs to know about chunks of data loaded, its method “DisplayProgress” can be registered with B to be invoked as a callback while the data is being loaded by B. Once the data has been loaded, it can be displayed using A’s method “DisplayData”. B offers a method “RegisterProgressCallback” with the attribute “CallbackRegistration” attached. Therefore, this method can be used to register component A’s method “DisplayProgress” as a callback. After that, B’s method “LoadData” can be called to initiate data loading. While the data is being loaded, the method will invoke the registered callback, which is illustrated by the attribute “IssueCallback” attached to the method.

The scenario required by the system developer seems to be fulfilled by assembling components A and B in this way. To confirm this, he can check the deployment contracts of A and Bin the manner described in the previous section.

However, Deployment Contracts Analysis finds out that component A has a component-level attribute “RequiredThreadAffineAccess” that requires all its methods to be called always from one and the same thread. The method “DisplayProgress” will be called from a thread internally created by the method “LoadData”. But the method “DisplayData” will be called from the main thread. This means that methods of A will be called from different threads, which contradicts its requirement that it requires thread-affine access. Furthermore, if data is loaded several times, the method “B.LoadData(...)” will create a new thread each time it is called thus invoking the method “A.DisplayProgress(...)” each time on a different thread. This means that A and B are incompatible.

A component from the assembly AB has to be replaced by another one. Then a deployment contracts analysis has to be performed again. This process has to be repeated until an assembly of compatible components, i.e. a conflict-free assembly, is found. Once a conflict-free assembly is found, it can be executed at runtime.

8 Evaluation

To evaluate our proposed component model with related approaches approaches we align them along the following criteria:

Metadata inside or outside component This criterion indicates whether a component has

metadata inside it or not. Some systems apply metadata to component specifications or put them into component descriptions. Other systems put metadata inside components themselves. With respect to deployment contracts, it is important to have metadata inside (binary) component because in this case it cannot be tampered with. However, it is equally essential to be able to retrieve the metadata at deployment time, i.e. before component instance creation.

Metadata used for some component behaviour analysis This criterion indicates whether metadata is used for an analysis (by a tool) in the system. If metadata is used for only communicating the container what to do with the component at runtime like in EJB or CCM it is not considered to be analysis. On the contrary, if metadata is used for deriving some information about component behaviour which is subsequently analysed to derive some more information about system behaviour, it is considered to be analysis.

Component lifecycle phase of metadata processing This criterion says whether metadata is processed at component deployment. In this criterion unlike in the previous one any evaluation of metadata is considered. Deployment contracts of components are analysed at component deployment time.

Metadata used for some component composition analysis This criterion indicates whether metadata of individual components is used to predict some behaviour of some composition of those components. In other words, this section points out whether some kind of predictable assembly of components by analysing component metadata is supported by the system. Deployment contracts are developed to be particularly suitable for this task.

Metadata specification method This criterion describes the method used to specify component metadata. Some systems use XML to specify metadata, others use special types or attributes. Deployment contracts are specified in a way that they are embedded into the (binary) component and can be retrieved without instance creation. Having metadata embedded into the component has the advantage that it cannot be easily tampered with. The ability to analyse metadata at deployment time without instance creation has the advantage that it can be retrieved before component can show any behaviour. When an instance of component is being created, the component can already show some behaviour, which has not been analysed before. Therefore, it is important to be able to analyse metadata before instance creation. Furthermore, sometimes a component instance cannot be created. The reasons for the failure cannot be discovered if a component instance is needed to retrieve metadata.

Table 7 shows that our approach is unique for the following reasons:

- Our pool of metadata for component deployment is general-purpose since it is created by analysing general-purpose comprehensive frameworks for component development. Other pools of metadata, see [29] for a survey, are not general-purpose but mostly specific to a domain: MetaH ADL has a set of metadata for component deployment in the domains of flight control and avionics; CR-RIO Framework has metadata for deployment of nodes of distributed applications in terms of distribution and processing policies.

- Use of metadata for component deployment in current component models [29] such as EJB and CCM is restricted to component deployment descriptors that are XML specifications

Property	*ACME Ext.*	*Comp. ModelExt.*	*EJB*	*CCM*	*.NET*	*Proposed Model*
Metadata in/outside component	*Outside*	*Inside*	*Outside*	*Outside*	*Inside*	*Inside*
Component behaviour analysis	*Yes*	*Yes*	*No*	*No*	*No*	*Yes*
Metadata processed@ component deployment	*No*	*No*	*No*	*No*	*Yes*	*Yes*
Metadata usedfor component composition analysis	*Yes*	*No*	*No*	*No*	*No*	*Yes*
Metadata specifi– cation method	*Annotations in component spec.*	*Attributes on comp.*	*XML comp. desc– riptors*	*XML comp. desc– riptors*	*Attributes on comp.*	*Attributes on comp.*

Table 7: A comparison of component models utilizing component metadata

describing how to manage components by the component container. Specification of metadata in an easily changeable form like XML has the disadvantage that it can be easily tampered with, which is fatal for system execution. Therefore, our metadata is contained in the real binary components, cannot be easily tampered with and is retrieved by the Deployment Contracts Analyser on component deployment.

- Moreover, metadata about components in deployment descriptors is not analysed for mutual compatibility, which deployment contracts analysis does. Although deployment descriptors allow specification of some component's environmental dependencies and some aspects of component's threading model, the information specifiable there is not comprehensive and only reflects features that are manageable by containers, which are limited. By contrast, our metadata set is comprehensive and the component developer is obliged to show all environmental dependencies and aspects of threading model for their component. In addition, on deployment contracts analysis we take into consideration properties of the system execution environment the component is deployed to as well as emergent assembly-specific properties, which other approaches do not do.

- Furthermore, in current component models employing metadata for component deployment, metadata is not analysed at component deployment time. For instance, in EJB and CCM the data in deployment descriptors is used by containers at runtime but not at deployment time. The deployment descriptor has to be produced at deployment time but its contents is used at runtime. In .NET, only metadata for graphical component arrangement is analysed at deployment time. By contrast, in our approach all the metadata is analysed at deployment time, which is essential with binary components from different

component suppliers.

- Additionally, using our attributes developers have extensive IDE support in form of IntelliSense. Moreover, .NET developers should be familiar with the concept of attributes thus making it easy for them to employ the proposed approach using new attributes. Thanks to various parameters on each attribute, the component developer can flexibly specify how resources are used inside component and which threading aspects are available.

Finally, the idea of Deployment Contracts based on a predefined pool of parameterisable attributes can be applied to any component model supporting composition of components at deployment time. We have implemented the idea in .NET and since .NET component model supports deployment time composition, our implementation is a direct extension of .NET component model with about 100 new attributes and a "deployment-time" analyser of them. However, the idea is general and therefore other frameworks for component development can be studied to create more attributes, thus enabling more comprehensive reasoning by extending deployment contracts analysis.

9 Conclusion

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proc. 16th European Conference on Object-Oriented Programming*, pages 334–367. Springer-Verlag, 2002.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.

[3] J. Aldrich, D. Garlan, B.R. Schmerl, and T. Tseng. Modeling and implementing software architecture with acme and archjava. In *Proc. OOPSLA Companion 2004*, pages 156–157, 2004.

[4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[5] D. Balek and F. Plasil. Software connectors: A hierarchical model. Technical Report Tech. Report No. 2000/2, Dep. of SW Engineering, Charles University, 2000.

[6] M.F. Bertoa and A. Vallecillo. Quality attributes for COTS components. *I+D Computación*, 1(2):128–144, November 2002.

[7] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

[8] COM web page. `http://www.microsoft.com/com/`.

[9] CORBA FAQ web page. `http://www.omg.org/gettingstarted/corbafaq.htm`.

[10] Microsoft Corporation. Microsoft asynchronous pattern for components.

[11] Microsoft Corporation. *Microsoft Visual Studio.NET*. Microsoft Press, 1st edition, 2001.

[12] Microsoft Corporation. Msdn .net framework class library version 2.0, 2005.

[13] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification Version 2.0*, 2001.

[14] Nimish Doshi. Object databases and multi-tier architectures. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 408, Washington, DC, USA, 1998. IEEE Computer Society.

[15] L. Dykes, E. Tittel, and C. Valentine. *XML Schemas*. Sybex Inc, 2002.

[16] R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.

[17] EZ JCom Framework web page. http://www.ezjcom.com.

[18] Martin Fowler, Don Box, Anders Hejlsberg, Alan Knight, Rob High, and John Crupi. The great j2ee vs. microsoft.net shootout. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 143–144, New York, NY, USA, 2004. ACM Press.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[20] D. Garlan, R. Monroe, and D. Wile. Acme: An architectural interconnection language. Technical Report CMU-CS-95-219, Carnegie Mellon University, 1995.

[21] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[22] Ian Gorton and Anna Liu. An architects guide to enterprise application integration with j2ee and .net. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 726–727, New York, NY, USA, 2005. ACM Press.

[23] Richard Monson Haefel. *Enterprise Java Beans*. O'Reilly, 4th edition, 2004.

[24] S.A. Hissam, G.A. Moreno, J.A. Stafford, and K.C. Wallnau. Packaging predictable assembly. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD 2002)*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag Heidelberg, Janaury 2002.

[25] Christine Hoffmeister, Dilip Soni, and Robert Nord. *Applied Software Architecture*, volume 99-4085 of *The Addison-Wesley object technology series*. Addison-Wesley Professional, 2000.

[26] J-Integra web page. http://j-integra.intrinsyc.com.

[27] Aaron W. Keen and Ronald A. Olsson. Exception handling during asynchronous method invocation. In *Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany*, volume 2400 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[28] Perrochon L. Translation servers: Gateways between stateless and stateful information systems. Report 1994PA-nsc94, Institut fur Informationssysteme, ETH Zurich, 1994.

[29] K.-K. Lau and V. Ukis. Component metadata in component-based software development: A survey. Preprint 34, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, October 2005.

[30] K.-K. Lau and V. Ukis. A container for automatic system control flow generation using exogenous connectors. Preprint 31, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, August 2005.

[31] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.

[32] Dennis Lee, Jean-Loup Baer, Brian Bershad, and Tom Anderson. Reducing startup latency in web and desktop applications. In *3rd USENIX Windows NT Symposium*, pages 165–176, Seattle, Washington, USA, July 1999.

[33] B. Lewis. Software portability gains realized with metah and ada95. In *Proceedings of the 11th international Workshop on Real-Time Ada Workshop*, 2002.

[34] B. Lewis, E. Colbert, and S. Vestal. Developing evolvable, embedded, time-critical systems with metah. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, 2000.

[35] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* Wiley, 2nd edition, February 2006.

[36] Craig Skibo Marc Young, Brian Johnson. *Inside Microsoft Visual Studio .NET 2003.* Microsoft Press, 2nd edition, 2003.

[37] V. Matena and B. Stearns. *Applying Enterprise JavaBeans – Component-based Development for the J2EE Platform.* Addison-Wesley, 2000.

[38] J. Matevska-Meyer, W. Hasselbring, and R.H. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *Proc. 9th Int. Workshop on Component-oriented Programming*, 2004.

[39] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proc. ACM SIGSOFT'96*, pages 24–32, 1996.

[40] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[41] Microsoft .NET web page. `http://www.microsoft.com/net/`.

[42] Sun Microsystems. Enterprise java beans specification, version 3.0, 2005.

[43] Mono – .NET CLR for Linux. `http://www.mono-project.com/About_Mono`.

[44] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 4th edition, 2004.

[45] Adam Nathan. *.NET and COM: The Complete Interoperability Guide*. Sams, 1st edition, 2002.

[46] .NET languages web page. http://www.gotdotnet.com/team/lang/.

[47] Eric Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Professional, 1st edition, May 2002.

[48] Johann Oberleitner and Michael Fischer. Improving composition support with lightweight metadata-based extensions of component models. In *Software Composition*, 2005.

[49] ObjectWeb – Open Source Middleware. *OpenCCM User's Guide*. `http://openccm.objectweb.org/doc/0.8.1/user_guide.html`.

[50] OMG. *UML 2.0 Superstructure Specification*. `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`.

[51] Object Management Group (OMG). Corba components, specification, version 0.9.0, 2005.

[52] D.S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, 2003.

[53] Ralf H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *Proceedings of the 34th Hawaii International Conference on System Sciences*. IEEE, 2001.

[54] E. Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform*. Wiley, enterprise edition, 1999.

[55] D. C. Schmidt, T. Harrison, and N. Pryce. Thread-specific storage - an object behavioral pattern for accessing per-thread state efficiently. In *The Pattern Languages of Programming Conference*, September 1997.

[56] Douglas C. Schmidt. *Pattern-oriented Software Architecture. Vol. 2, Patterns for Concurrent and Networked Objects*. New York John Wiley&Sons, Ltd., 2000.

[57] J. A. Stafford and A. L. Wolf. Annotating components to support component-based static analyses of software systems. In *Procoeedigns of the Grace Hopper Celeb. of Women in Computing*, 2001.

[58] Sun Microsystems. *The Bean Builder*. `https://bean-builder.dev.java.net/`.

[59] Sun Microsystems. *Java 2 Platform, Enterprise Edition*. `http://java.sun.com/j2ee/`.

[60] Sun Microsystems. *JavaBeans Architecture: BDK Download*. `http://java.sun.com/products/javabeans/software/bdk_download.html`.

[61] Sun Microsystems. *JavaBeans Specification*, 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.

[62] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[63] T. Valesky and T.C. Valesky. *Enterprise JavaBeans: Developing Component-Based Distributed Applications*. Addison-Wesley, 1999.

[64] G. Venkitachalam and Tzicker Chiueh. High performance common gateway interface invocation. In *IEEE Workshop on Internet Applications*, pages 4–11, San Jose, CA, USA, August 1999.

[65] A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright. *Microsoft .NET Compact Framework(Core Reference)*. Microsoft Press, January 2003.

[66] Joseph Williams. The web services debate: J2ee vs. .net. *Commun. ACM*, 46(6):58–63, 2003.

[67] Ron Zahavi. *Enterprise Application Integration with CORBA Component and Web-Based Solutions*. John Wiley and Sons, 1st edition, 1999.