Modern Software Architectures: Novel Solutions or Old Hats?

Petr TUMA

Dept. of Software Engineering, Charles University
Malostranské nám. 25, 118 00 Praha, Czech Republic
petr.tuma@mff.cuni.cz

Abstract. In the increasingly complex domain of information systems, an important role is attributed to their architecture. The developers are offered concepts such as component or multi-tier architectures and tools to implement them such as the application servers or web services. These are often a mixture of proven and novel solutions, whose overview and evolution from the recent past to the future trends is outlined in this article.

Keywords: software architecture, component architecture, three-tier architecture, application server, web service.

1 Introduction

Ever since the inception of the software engineering discipline, an important role was attributed to the architecture of a software system, or a software architecture for short. Defined as "a set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces (...) together with their behavior (...), the composition of these structural and behavioral elements (...) and the architectural style that guides this organization" [4], the software architecture influences both the development process and the properties of the developed system. To achieve better control over these, software engineering is therefore seeking proven solutions at the level of software architecture.

The goal of this paper is to present the component and multi-tier architectures as two prominent solutions at the level of software architecture, together with the application servers and web services as two tools to implement such solutions. The paper also strives to avoid the marketing slant often associated with these architectures and tools due to their current popularity, and separate and explain their proven and novel aspects by following their development from the recent past to the future trends.

The paper is structured around the two architectures and the two tools presented, each being dedicated a separate section. Each of these four sections starts with an overview of its target architecture or tool, explaining its purpose and aims. Following the overview, separate subsections deal with outlining the benefits and the pitfalls involved. The last part of each section sketches the future trends. The conclusion and references follow as usual.

2 Presented Architectures

2.1 Component Architectures

The concept of component architectures extends the idea that a software system can be assembled from available building blocks rather than built entirely from scratch. Historically, this idea reaches back to the late 1960s and 1970s, when the concepts of modularization and information hiding first emerged [18]. It is also present in many concepts widely accepted and used today, most notably in libraries and frameworks. What distinguishes the component architectures from these similar concepts are the properties of the building blocks used to assemble a software system.

In the component architectures, the building blocks are typically called software components. Of the many definitions of a software component, an often quoted one states that a software component is "a unit of composition with contractually specified interfaces and explicit context dependencies only (…) that can be deployed independently and is subject to composition by third parties" [28]. This definition highlights four important properties of a component, which are the specified interfaces, the explicit dependencies, the independent deployment and the third party composition.

The fact that a software component has contractually specified interfaces and only explicit dependencies implies that all interaction between the component and the software system that uses it happens only through its interfaces, and that the software system can only rely on what the interfaces of the component specify. This makes it possible to use the component as a black box, with no knowledge of its internals.

In a similar manner, the fact that a software component can be deployed independently and composed by third parties implies that the use of a component as a black box extends beyond a single software system. The party developing the software system can differ from the party developing the component, and the two parties can also differ from the party deploying the components into the software system.

The interaction of multiple parties in developing a software system requires that the outlined properties of a software component are not only present at the conceptual level, but also compatible at the technical level. This is achieved by standardizing a component model, which defines the properties of a software component down to the technical details of how the interfaces of a component are specified or how a component is deployed. Examples of component models include the Component Object Model (COM) from Microsoft [12], the CORBA Component Model (CCM) from OMG [15] and others. A brief overview of CCM is provided for illustration.

The first major part of the CCM standard is an interface definition language for the specification of the component interfaces. The language distinguishes facets, receptacles, event sources, event sinks and attributes as five basic types of ports that a component can use to interact with the software system that uses it. Using a general programming terminology, facets describe functions exported by the component, receptacles describe functions imported by the component, event sources describe messages sent by the component, event sinks describe messages received by the component, and attributes describe variables exported by the component. An example of an interface specification is on figure 1.

```
interface SomeInterface
{
  void SomeMethod (in short some_argument);
}

valuetype SomeEvent : Components::EventBase
{
  public string some_content;
}

component SomeComponent
{
  provides SomeInterface provided_interface;
  requires SomeInterface required_interface;
  publishes SomeEvent published_event;
  consumes SomeEvent consumed_event;
}
```

Fig. 1. CCM Interface Definition Language

The second major part of the CCM standard is the component implementation framework, which describes how to implement a component given a specification of its interfaces. The framework introduces a set of language mappings for major implementation languages such as C++ and Java, with each language mapping explaining how to translate an interface specification from the interface definition language into the particular implementation language. Once the interface specification is translated, the task of implementing a component is basically the same as any other task of implementing a set of functions given their signatures.

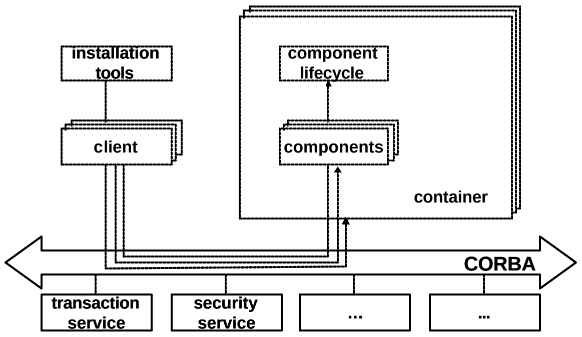

Fig. 2. CCM Architecture

The third major part of the CCM standard is a specification of the deployment and runtime environment available to the components. This includes guidelines for packaging the components so that they can be deployed into any compliant software system, and an architecture of the services provided to the components at runtime. This architecture is depicted on figure 2.

2.1.1 Benefits

The analyses of the benefits offered by the component architectures tend to focus on their most visible feature, which is the ability to reuse components. The alleged benefits associated with reuse are many, foremost among them the increase in productivity owed to the fact that whatever is available as a component does not have to be developed again. Also quoted are the increases in quality and speed of software systems, attributed to the fact that a fix or an optimization of a component impacts all the systems that use it, and the increases in interoperability [20].

It is important to note, however, that the cited benefits are associated with reuse in general rather than components in particular. The benefits of reuse are not a novel contribution of the component architectures, as they are already provided for example by libraries and frameworks. Also important is the fact that so far, reuse was only working well when used in narrow problem domains. There is no reason to expect the component architectures to change that.

Besides reuse, an important feature of the component architectures is the fact that a component can be deployed into a software system that uses it later than when this system is compiled. The deployment, or the linking of the component and the software system, can be postponed to when the system is being installed, or even when the system is being run. This brings the important benefit of being able to design a software system whose features can be later augmented by adding new components. This benefit is already used by many software systems to cope with data formats that the systems were not originally designed to handle, such as in the case of office suites manipulating external data from other programs, or in the case of web browsers displaying pages with embedded multimedia content.

2.1.2 Pitfalls

As was the case with benefits, many pitfalls of the component architectures are also associated with the ability to reuse components. The use of components requires an extra initial investment during the development of a software system, because the suitable components have to be selected and purchased. Other investments might be associated with the training necessary for using the components. How this extra cost balances with the benefits of reuse depends on the exact circumstances and is difficult to predict.

A potentially more dangerous problem, however, is associated with the need for maintenance of the software system. Given that the party developing the software system differs from the party developing the components, it is realistic to expect that the two parties will observe different priorities. This can result in the component developer neglecting to fix bugs that the software system developer encounters, or in the component developer deciding to introduce changes that the software system developer finds detrimental. This situation is analyzed in detail in [10].

2.1.3 Future trends

While the technical details of the component architectures, such as how the interfaces of a component are specified or how a component is deployed, appear to be solved, open issues still remain where the use of a component as a black box is concerned. It would be naïve to believe that a component can be used with absolutely no knowledge of its internals, as that would make it impossible to select proper components for use in a software system or guarantee important properties of the system such as resource requirements, performance or scalability. To remedy this, formalisms are being devised to describe behavior and other properties of a component [9, 11, 13, 19]. This description is likely to become a part of the specification of the component interfaces.

A related trend is associated with the attractiveness of components as building blocks of a software system. The components appear to be useful for a hierarchical decomposition of software architectures, and the latest developments in the area of UML [16] and associated methodologies such as Catalysis [6] support this use. The hierarchical decomposition, however, puts a very different emphasis on the properties of a component than the use described here. Because of this difference, it is maybe better to understand the components in the two contexts as two different concepts, rather than a single concept with two uses.

2.2 Multi-Tier Architectures

The concept of multi-tier architectures traces its origins back into the late 1980s, when the local area networks emerged as an affordable technology. This opened the possibility to exploit networks of computers where standalone systems were used before, and gave rise to the client-server or two-tier architecture. A software system that adheres to this architecture is split into two distinct parts, a client tier and a server tier, communicating through the network.

Assume an information system that consists of a database to store the information, a business logic to manipulate the information, and a user interface to present the information to the users, built using a two-tier architecture. An obvious position for the database in the architecture is the server tier, which is in a unique position to coordinate information sharing, and can reside on a powerful computer that meets the resource requirements typically associated with a database. Similarly, an obvious position for the user interface is the client tier. Unfortunately, no obvious place exists in the architecture for the business logic, which tends to be split arbitrarily between the client and the server tiers.

The lack of a clear place for the business logic in a two-tier architecture yielded two variants of the architecture, typically referred to as thin-client and thick-client variants. A thin-client two-tier architecture places most of the business logic in the server tier, minimizing the complexity of the client tier and thus also reducing the requirements on the computers hosting the client tier and the associated installation and maintenance costs. A thick-client two-tier architecture places most of the business logic in the client tier, minimizing the complexity of the server tier and thus also reducing the associated development and maintenance costs.

The attempts at avoiding the compromises associated with choosing either the thin-client or the thick-client variant of the two-tier architecture lead to the addition of another tier, a middle tier, forming a three-tier architecture. The three-tier architecture places the database in the server tier, the user interface in the client tier, and the business logic in the middle tier, as outlined in an example on figure 3.

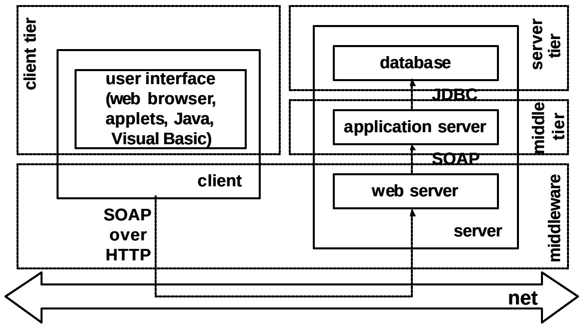

Fig. 3. Three-Tier Architecture

Naturally, with breaking the stereotype of having only two tiers comes a proliferation of additional tiers. The contemporary multi-tier architectures may thus distinguish a user tier for authentication and authorization, a workspace tier for establishing sessions, an enterprise tier for the business logic, or a resource tier for sharing resources. What remains as a distinguishing characteristic is the separation of the client and the server tier from the business logic.

2.2.1 Benefits

The principal benefit of multi-tier architectures is the degree of standardization of software architectures that makes it possible to develop a standardized supporting infrastructure. In the client tier, an example of such an infrastructure are the web browsers with applet and scripting abilities, used to implement a user interface that is easy to install and maintain on stock computers. In the middle tier, an example of such an infrastructure are the application servers with lifecycle and transaction support, used to implement a business logic that is easy to integrate with the user interface and the database. The infrastructure also includes the communication middleware, used to facilitate network communication through remote procedure calls, messages or streams.

2.2.2 Pitfalls

The general complexity related to the existence of multiple tiers, as well as the associated communication overhead, can have a negative impact on a software system. This impact, however, is obviously inherent to the multi-tier architectures, and thus probably better interpreted as a feature rather than a pitfall.

A less visible problem of the multi-tier architectures is the tendency of the supporting infrastructure to get bloated with features. The pressure to provide maximum functionality and interoperability leads to packages that offer integrated solutions for communication, persistency, replication, transactions, security and management, often with a choice of multiple protocols and services for compatibility with existing standards. The resource requirements of the many features of the infrastructure add up to a degree that may become unjustifiable, especially in software systems that do not use all of the features.

2.2.3 Future Trends

The lifecycle of an information system frequently includes integration with other information systems. For the purpose of integration, multi-tier architectures may be too restrictive if the individual tiers are coupled too tightly. This results in a trend to couple the tiers loosely, using dynamic binding at deployment time or even dynamic location and binding at runtime. Following the same trend are the attempts to supplement multi-tier architectures with a concept of independent communicating agents, but it remains to be seen how delivering a specific functionality is to be guaranteed with as little control over the agents as the concept allows.

3 Presented Tools

3.1 Application Servers

The application servers are a part of the infrastructure to support multi-tier architectures, especially where the tiers with the business logic are concerned. Historically, application servers are close to transaction monitors [8], with most of their concepts originating in 1980s. An application server provides a mechanism to handle the communication with the client tier, based on either the remote procedure call or message queuing paradigm. Besides communication, an application server usually provides database access by offering object persistency and distributed transactions. Various means, such as resource pooling and replication, are employed to achieve scalability.

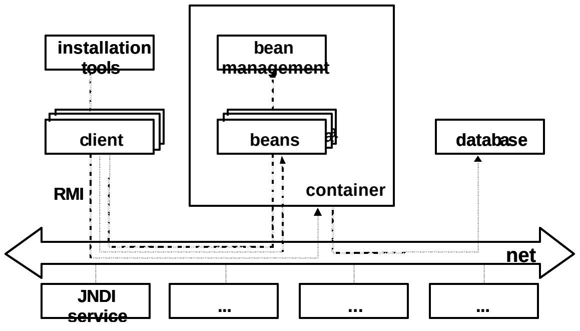

Fig. 4. EJB Architecture

Today, most application servers support the Enterprise Java Beans (EJB) standard from Sun Microsystems [24]. The EJB standard assumes a three-tier architecture, displayed on figure 4. The middle tier is conceived as a container for beans, the standard term for components that implement the business logic of an information system. The container controls the lifecycle of the beans and provides both declarative and procedural means to control the persistency of the bean state and the transactional character of the bean operations. Additional standards are employed by EJB, such as JDBC [25], JTA [27] for interaction with the server tier, or RMI and JNDI [26] for communication with the client tier.

3.1.1 Benefits And Pitfalls

As an infrastructure to support multi-tier architectures, the application servers both provide the benefits and suffer the pitfalls that were already outlined in sections 2.2.1 and 2.2.2.

3.1.2 Future Trends

The development of the application servers is conservative in nature, owing to the fact that the application servers are commercial products rather than research projects. Evident is a trend to structure the application server as a collection of building blocks rather than an integrated package, with technologies such as aspects [2] and reflection [21] explored to achieve this trend.

3.2 Web Services

The web services standardize an infrastructure for integrating information systems in the environment of the Internet. The web services are based on the Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) standards.

The SOAP standard [31] by W3C defines a communication protocol based on a textual form of message encoding in XML [30]. Each message consists of a series of optional headers and a body, with the headers carrying information intended for systems that route the message and the body intended for the final recipient of the message. The mes sages are extensible and easy to transport via HTTP [7].

The WSDL standard [32] by W3C defines a web service description in XML. For each service, the description specifies all the data types and mess age formats used by the service, the message encoding and the communication protocol supported by the service, and the network addresses of the service. The description thus provides all the information that is required to set up communication with the service.

The UDDI standard [29] by the UDDI Consortium defines a web service capable of registering and locating other web services. UDDI relies on the industry classification standards, such as NAICS or UNSPSC, to provide a hierar chy to sort the services by. For each service, UDDI records its position in the hierarchy together with an information about its provider and its WSDL description.

The interaction of the SOAP, WSDL and UDDI standards is outlined on figure 5. When in need of a specific service, a client first uses the UDDI service to locate the required service by its classification and obtain its WSDL description. The WSDL description is then used to set up communication with the required service, typically using the SOAP protocol transported via HTTP.

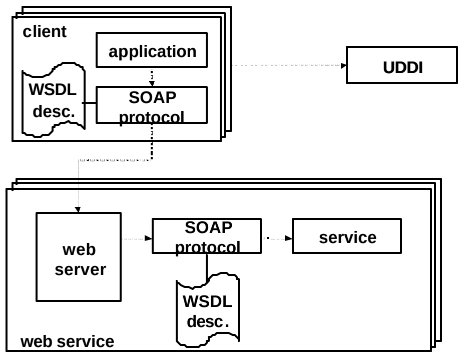

Fig. 5. Web Services Architecture

3.2.1 Benefits

The benefit of the web services rests with providing an implementation of proven concepts specialized for the environment of the Internet. Compared with earlier implementations of the remote procedure call paradigm, such as Sun RPC [22] or OMG CORBA [14], the SOAP protocol works well in presence of firewalls, whose focus on the HTTP protocol is often detrimental to other network traffic. Similarly, the WSDL description does not require the compilation and linking, which is typically necessary when using descriptions in languages such as Sun XDR [23] or OMG IDL [14].

Considered individually, these benefits may not seem very important. Added together, however, they make the web services an infrastructure that is decidedly easier to use in the environment of the Internet than the earlier technologies.

3.2.2 Pitfalls

As an infrastructure for integrating information systems, the web services solve many problems associated with the technical side of communication, but say nothing about the semantic meaning of communication. Given that semantic incompatibility can present a serious obstacle to integration, questions need to be asked especially about the ability of the UDDI service to locate a required service by its classification only.

Also questioned is the overhead associated with textual form of message encoding. For all but textual information, the message encoding is inefficient and implies increased resource requirements during both transport and processing. These and other pitfalls of web services are further detailed in [1].

3.2.3 Future Trends

As the web services become prominent, new standards are being developed to complement SOAP, WSDL and UDDI where the technical side of communication is concerned. These include the Security Assertion Markup Language (SAML) for authentication and authorization, the Business Transaction Protocol (BTP) for business transactions, the Web Services Coordination (WS-Coordination), Web Services Transaction (WS-Transaction) and Business Process Execution Language for Web Services (BPEL4WS) standards for business process management, the Web Services Component Model (WSCM) for components, and others, outlined in more detail in [17]. Separately, standards such as EDI and SWIFT are being extended to support web services and thus address the semantic meaning of communication in the business domain [5].

Overall, the progress of web services makes the concept of independent communicating agents from section 2.2.3 more viable by providing an infrastructure, which encourages thinking along the lines of the concept, even if not solving all the related problems.

4 Conclusion

The paper has presented the component and multi-tier architectures as two prominent solutions at the level of software architecture, debating the benefits and pitfalls of both architectures. The paper has also described the application servers and web services as two tools to implement both architectures, again debating the benefits and pitfalls. Avoiding the marketing slant often associated with these architectures and tools, it can be concluded that they represent an evolutionary rather than a revolutionary step, with majority of the concepts existing since 1980s. That being said, however, does not diminish the usefulness of these architectures and tools, nor their real contribution as outlined in the paper.

5 Acknowledgements

Thanks go to the members of the Distributed Systems Research Group of Charles University, Prague, for valuable comments on this article. The work on this article was partially sponsored by the Grant Agency of the Czech Republic grant number 201/03/0911.

6 References

1. Alonso, G.: Myths Around Web Services. *IEEE Data Engineering Bulletin,* Vol. 25 No. 4, IEEE, 2002.
2. Aspect-Oriented Software Association: *Aspect-Oriented Software Development.* http://www.aosd.net.
3. Barroca, L., Hall, J., Hall, P.A.: *Software Architecture: Advances and Application.* Springer-Verlag, 1999.
4. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
5. Bussler, C.: B2B Protocol Standards and their Role in Semantic B2B Integration Engines. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering,* Vol. 24 No. 1, IEEE , 2001.
6. D'Souza, D.F., Wills, A.C.: *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1999.
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: *Hypertext Transfer Protocol.* IETF RFC 2616, 1999.
8. Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.
9. Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice Hall, 1985.
10. Lehman, M.M., Ramil, J.F.: Software evolution in the age of component-based software engineering. *IEE Proceedings Software,* Vol. 147 No. 6, IEE, 2000.
11. Manna, Z., Pnueli, A.: *Temporal Logic of Reactive Systems: Specification.* Springer-Verlag, 1992.
12. Microsoft: *Component Object Model.* http://www.microsoft.com/com.
13. Milner, R.: *Calculus of Communicating Systems.* Springer-Verlag, 1982.
14. Object Management Group: *Common Object Request Broker Architecture.* http://www.omg.org.
15. Object Management Group: *CORBA Component Model.* http://www.omg.org.
16. Object Management Group: *Unified Modeling Language.* http://www.uml.org.
17. Ogbuji, U.: *The Past, Present and Future of Web Services.* http://www.webservices.org.
18. Parnas, D.: On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM,* Vol. 15 No. 12, ACM, 1972.
19. Plášil, F., Višnovský, S.: Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering,* Vol. 28 No. 11, IEEE, 2002.
20. Sametinger, J.: *Software Engineering with Reusable Components.* Springer-Verlag, 1997.
21. Smith, B.C.: *Procedural Reflection in Progr amming Languages.* MIT, 1982.

22. Srinivasan, R.: *RPC: Remote Procedure Call Protocol Specification Version 2.* IETF RFC 1831, 1995.
23. Srinivasan, R.: *XDR: External Data Representation Standard.* IETF RFC 1832, 1995.
24. Sun Microsystems: *Enterprise Java Beans.* http:// java.sun.com/products/ejb.
25. Sun Microsystems: *Java Database Connectivity API.* http://java.sun.com/products/jdbc.
26. Sun Microsystems: *Java Naming and Directory Interface.* http://java.sun.com/products/jndi.
27. Sun Microsystems: *Java Transaction API.* http://java.sun.com/products/jta.
28. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
29. UDDI Consortium: *Universal Description, Discovery and Integration.* http://www.uddi.org/specification.html.
30. W3C: *Extensible Markup Language.* http://www.w3.org/TR.
31. W3C: *Simple Object Access Protocol.* http://www.w3.org/TR.
32. W3C: *Web Services Description Language.* http://www.w3.org/TR.