Automated Testing of Server-Side Software Components

BONGIOVANNI FRANCESCO

Master of Science Thesis
Stockholm, Sweden, 2008

ICT/ECS-2008-97
Automated Testing of Server-Side Software Components

BONGIOVANNI FRANCESCO

Supervisor and Examiner:

Dr. Vladimir Vlassov

Department of Electronic, Computer and Software Systems (ECS), School of Information and Communication Technology (ICT) KTH-Royal Institute of Technology
“Computer science is no more about computers than astronomy is about telescopes.”

- Edsger Dijkstra
Abstract

Web services are one of the latest distributed systems paradigms based on distributed components used in many enterprises for creating large scale distributed enterprise applications. Unique aspects such as late-binding, asynchronous messaging, composition of remote Web services owned by different companies make Web Services testing complex, costly and sometimes infeasible due to the exponential number of possible combinations. Automating the process of testing such components is critical.

This thesis presents concepts from the software testing area, the various types of testing, as well as several metrics for measuring the quality of software. It will question the need for automating the testing process by discussing manual and automated tests.

It discusses server-side components technologies, with a strong emphasis on Web Services, along with some of its related specifications such as SOAP and WS-BPEL 2.0. It discusses some methods for testing Web Services in isolation and composite Web Services.

It will also present a case study which introduces a testing ecosystem I proposed while working at Amadeus IT Group for testing parts of their server-side components. Their testing process is based on the design and the execution of specific test scenario. In my solution, these scenarios are created and executed using BPEL processes.

This testing ecosystem is tightly linked to Web Services due to the recursive nature of a BPEL process, that is, when deployed a BPEL process becomes a Web Service and due to the future nature of the components. This ecosystem relies on various components which, when combined, provides a flexible environment for creating and executing tests.

Some prototypes were developed as proof of concepts after the internship and aim at enhancing the ecosystem by providing new capabilities, such as the possibility to remotely run some tests from any machine which exposed them with the help of a specific Web Service engine, Axis2.

Keywords

Testing, Web Services, SOAP, BPEL, Apache Axis2, composite Web Services, distributed systems.
Acknowledgements

This thesis is the final work of the Master’s of Science in Software Engineering of Distributed Systems programme (SEDS) at the Royal Institute of Technology (KTH). The thesis partially bases its ideas on a 6 months internship done at Amadeus IT Group in Nice, France.

First I would like to thank Amadeus and in particular Loïc Pierlot, for giving me the opportunity to do an internship within a great team in such a nice company. I would like to thank my former co-workers: Elona, Riadh, Gerry, Cécile, Jerome, Tadhg, Guillaume and all the people at Amadeus for our pleasant discussions and great coffee times we shared altogether.

I am also thankful to my supervisor Dr Vladimir Vlassov from KTH, for his precious advices he gave me when I needed them the most, and for his patience. I thank him a million times for the freedom he gave me for working on this thesis. I would like also to thank all my KTH teachers, this university is a great place and changes your mind for the better.

A special thanks to former interns and colleagues who worked in the same office as I did: Fatima, Elhadji, Julien, Jichuan, Thomas and Yann for providing me a friendly work environment.

Thanks to my SEDS friends, Eneko, Hasnain and Truong for their invaluable friendship and the many nice times we had at Forum, while working and/or studying the whole night long.

A big thanks to my former room-mate, Camilo, it was a nice experience living with you, to Sanjay, both Ana’s, Loida, Christian, Natalia, Gaby, Sarah…it was really nice meeting all of you!

Last but not least, I wish to thank my parents, for their unconditional support throughout my life, for letting me do my own mistakes and be there to help me when I needed it. I thank my sisters, Sikelia for her own special way of cheering me up, my younger sister Nathalia for her support, kindness, patience when listening to me when I am speaking about geeky stuffs and for encouraging me to follow my passion. I finally thank my little brother Anthony for reminding me that a computer is not only a machine to work on, but also a machine to play with.

Namur (Belgium), September 2008
Francesco Bongiovanni
# Table of Contents

Abstract ................................................................................................................................................... 1  
Table of Figures ....................................................................................................................................... 6  
Abbreviations and Acronyms .................................................................................................................. 8  
Introduction ............................................................................................................................................. 9  
  1.1. Problem statement ........................................................................................................................... 9  
  1.2. Motivation ......................................................................................................................................... 10  
  1.3. Goals and expected results .............................................................................................................. 10  
  1.4. Thesis structure and outlines ........................................................................................................... 11  
Part 1 ..................................................................................................................................................... 14  
  1. Introduction to software testing .......................................................................................................... 15  
    1.1. Software failures ........................................................................................................................... 15  
    1.2. Software testing ............................................................................................................................ 17  
    1.3. What type of software testing does exist? ....................................................................................... 19  
    1.3.1. Objectives of testing .................................................................................................................. 19  
    1.3.2. Unit testing ................................................................................................................................. 19  
    1.3.3. Integration testing ...................................................................................................................... 19  
    1.3.4. System testing ............................................................................................................................ 20  
    1.3.5. Other types of testing .................................................................................................................. 20  
    1.4. Measuring the quality .................................................................................................................... 22  
    1.4.1. Black box testing methods ........................................................................................................ 22  
    1.4.2. White box testing metrics .......................................................................................................... 25  
    1.4.3. Which metric to choose? .............................................................................................................. 31  
    1.5. Challenges in today's software testing ......................................................................................... 32  
  2. Server side components ....................................................................................................................... 33  
    2.1. Software components ..................................................................................................................... 33  
    2.1.1. History and goals of software componentization .................................................................... 34  
    2.2. Server-side components ................................................................................................................ 35  
    2.2.1. CORBA ...................................................................................................................................... 36  
    2.2.2. DCOM ...................................................................................................................................... 37  
    2.2.3. Java RMI ................................................................................................................................... 37  
    2.2.4. Web Services ............................................................................................................................. 39  
    2.2.5. Differences between components and Web services .................................................................. 47  
  3. Web Service Compositions .................................................................................................................. 49  
    3.1. WS-BPEL ...................................................................................................................................... 49  
    3.1.1. Initial example: the purchase order process .............................................................................. 52  
    3.1.2. WS-BPEL 2.0 process overview ............................................................................................... 53
<table>
<thead>
<tr>
<th>4</th>
<th>Test automation .................................................................</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Goals of test automation ...............................................</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Manual testing and automated testing ................................</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>When should we automate tests? ........................................</td>
<td>62</td>
</tr>
<tr>
<td>5</td>
<td>Testing Web Services and composite Web Services ..................</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Unit testing ........................................................................</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>Integration testing ..........................................................</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Related work .......................................................................</td>
<td>78</td>
</tr>
<tr>
<td>Part 2</td>
<td>.........................................................................................</td>
<td>82</td>
</tr>
<tr>
<td>6</td>
<td>Case study: Shopping Dossier Automated Tester (SDAT) ............</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>Background ..........................................................................</td>
<td>83</td>
</tr>
<tr>
<td>6.2</td>
<td>Problem definition ...........................................................</td>
<td>83</td>
</tr>
<tr>
<td>6.3</td>
<td>Objectives and requirements of the tool ..............................</td>
<td>84</td>
</tr>
<tr>
<td>6.4</td>
<td>Expectations .......................................................................</td>
<td>85</td>
</tr>
<tr>
<td>6.5</td>
<td>Shopping Dossier general architecture ...............................</td>
<td>87</td>
</tr>
<tr>
<td>7</td>
<td>Case study (SDAT): Investigations and solution proposal ........</td>
<td>88</td>
</tr>
<tr>
<td>7.1</td>
<td>Creating the scenarios ......................................................</td>
<td>88</td>
</tr>
<tr>
<td>7.1.1</td>
<td>BPMN or BPEL? ..................................................................</td>
<td>88</td>
</tr>
<tr>
<td>7.2</td>
<td>Executing the scenarios ....................................................</td>
<td>92</td>
</tr>
<tr>
<td>7.3</td>
<td>Web Service Testing Tool ..................................................</td>
<td>94</td>
</tr>
<tr>
<td>7.4</td>
<td>Issues ahead .......................................................................</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>Prototypes architecture and implementation ........................</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>SoapUI Extension .............................................................</td>
<td>100</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Purpose .............................................................................</td>
<td>100</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Architecture .....................................................................</td>
<td>100</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Implementation details .....................................................</td>
<td>101</td>
</tr>
<tr>
<td>8.2</td>
<td>BPEL Monitoring Web UI and SoapWUI ...............................</td>
<td>103</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Purpose .............................................................................</td>
<td>103</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Architecture .....................................................................</td>
<td>104</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Implementation details .....................................................</td>
<td>105</td>
</tr>
<tr>
<td>9</td>
<td>Evaluation and conclusion ..................................................</td>
<td>108</td>
</tr>
<tr>
<td>9.1</td>
<td>Evaluation .................................................................</td>
<td>108</td>
</tr>
<tr>
<td>9.2</td>
<td>Testing ............................................................................</td>
<td>109</td>
</tr>
<tr>
<td>9.3</td>
<td>Conclusion and future work .............................................</td>
<td>114</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Suggestions for future work and enhancements ..................</td>
<td>115</td>
</tr>
<tr>
<td>Bibliography .................................................................</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>Appendix A .................................................................</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>Notorious software failures .............................................</td>
<td>121</td>
</tr>
</tbody>
</table>
A.2 The minefield paradox ........................................................................................................ 122
Appendix B .................................................................................................................................. 124
B.1 WSDL 2.0 example ............................................................................................................. 124
B.2 SOAP encoding styles ......................................................................................................... 127
B.3 Detailed explanation of the BPEL language constructs ...................................................... 132
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1:</td>
<td>How to understand the thesis' title</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2:</td>
<td>Basic cause-effect graphs</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3:</td>
<td>White box testing related metrics</td>
<td>25</td>
</tr>
<tr>
<td>Figure 4:</td>
<td>Small program to be tested, visual representation and Java code (courtesy of G. Myers, redrawn)</td>
<td>26</td>
</tr>
<tr>
<td>Figure 5:</td>
<td>Definition and use in data flow testing</td>
<td>27</td>
</tr>
<tr>
<td>Figure 6:</td>
<td>Finite State Machine example</td>
<td>29</td>
</tr>
<tr>
<td>Figure 7:</td>
<td>Another Finite State Machine example</td>
<td>30</td>
</tr>
<tr>
<td>Figure 8:</td>
<td>An 'Authorization' component which can be used for security purposes.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9:</td>
<td>In red (login,register) you can see the interfaces that the 'Authorization' component provides, and in blue the dependency it needs (CheckIfUserExist)</td>
<td>33</td>
</tr>
<tr>
<td>Figure 10:</td>
<td>Components' history evolution. (Diagram based on lectures notes of Martin Henkel (lecturer at KTH) in Model Driven Architecture of Components course)</td>
<td>35</td>
</tr>
<tr>
<td>Figure 11:</td>
<td>Typical client/server architectural style, where the cloud represents the Internet</td>
<td>36</td>
</tr>
<tr>
<td>Figure 12:</td>
<td>A client sending a request to an object through the ORB</td>
<td>36</td>
</tr>
<tr>
<td>Figure 13:</td>
<td>Stub/skeleton concept in a distributed environment</td>
<td>38</td>
</tr>
<tr>
<td>Figure 14:</td>
<td>Service oriented architecture</td>
<td>39</td>
</tr>
<tr>
<td>Figure 15:</td>
<td>Web Services architecture with its core specifications: UDDI, SOAP and WSDL</td>
<td>40</td>
</tr>
<tr>
<td>Figure 16:</td>
<td>WDSL 2.0 general structure</td>
<td>41</td>
</tr>
<tr>
<td>Figure 17:</td>
<td>WSDL 1.1 general structure</td>
<td>42</td>
</tr>
<tr>
<td>Figure 18:</td>
<td>SOAP message layout</td>
<td>44</td>
</tr>
<tr>
<td>Figure 19:</td>
<td>SOAP request example</td>
<td>45</td>
</tr>
<tr>
<td>Figure 20:</td>
<td>Web Services stack</td>
<td>46</td>
</tr>
<tr>
<td>Figure 21:</td>
<td>The need for Web service composition</td>
<td>49</td>
</tr>
<tr>
<td>Figure 22:</td>
<td>Using BPEL to create a composite Web Service</td>
<td>50</td>
</tr>
<tr>
<td>Figure 23:</td>
<td>WS-BPEL and related specifications</td>
<td>51</td>
</tr>
<tr>
<td>Figure 24:</td>
<td>Purchase Order (PO) process example</td>
<td>52</td>
</tr>
<tr>
<td>Figure 25:</td>
<td>Higher level view of the process</td>
<td>53</td>
</tr>
<tr>
<td>Figure 26:</td>
<td>WSDL interface of the purchaseOrder</td>
<td>54</td>
</tr>
<tr>
<td>Figure 27:</td>
<td>Graphical BPEL representation of the purchaseOrder process</td>
<td>55</td>
</tr>
<tr>
<td>Figure 28:</td>
<td>XML representation of the graphical purchaseOrder BPEL process</td>
<td>57</td>
</tr>
<tr>
<td>Figure 29:</td>
<td>Steps involved in a test automation infrastructure</td>
<td>59</td>
</tr>
<tr>
<td>Figure 30:</td>
<td>Results of a GOOD test automation strategy. (Courtesy of Gerard Meszaros, used with permission)</td>
<td>60</td>
</tr>
<tr>
<td>Figure 31:</td>
<td>Results of a BAD test automation strategy. (Courtesy of Gerard Meszaros, used with permission)</td>
<td>60</td>
</tr>
<tr>
<td>Figure 32:</td>
<td>Application based on 'reliable' services</td>
<td>64</td>
</tr>
<tr>
<td>Figure 33:</td>
<td>Source code mutation example</td>
<td>69</td>
</tr>
<tr>
<td>Figure 34:</td>
<td>WSDL 1.1 Over-simplified UML metamodel</td>
<td>70</td>
</tr>
<tr>
<td>Figure 35:</td>
<td>Two equivalent XML documents</td>
<td>71</td>
</tr>
<tr>
<td>Figure 36:</td>
<td>Four levels WSDL-Based Test Case generation proposed by authors in (30)</td>
<td>73</td>
</tr>
<tr>
<td>Figure 37:</td>
<td>Example of a double datatype with a pattern constraint</td>
<td>74</td>
</tr>
<tr>
<td>Figure 38:</td>
<td>BPEL process with several paths</td>
<td>77</td>
</tr>
<tr>
<td>Figure 39:</td>
<td>Target and Default Exchange mutation operator</td>
<td>80</td>
</tr>
<tr>
<td>Figure 40:</td>
<td>Element Namespace Replacement mutation operator</td>
<td>80</td>
</tr>
<tr>
<td>Figure 41:</td>
<td>ComplexType Compositors Replacement mutation operator</td>
<td>80</td>
</tr>
<tr>
<td>Figure 42:</td>
<td>ComplexType Order Change mutation operator</td>
<td>80</td>
</tr>
<tr>
<td>Figure 43:</td>
<td>Test scenario example</td>
<td>84</td>
</tr>
<tr>
<td>Figure 44:</td>
<td>Example of scenarios flow</td>
<td>85</td>
</tr>
<tr>
<td>Figure 45:</td>
<td>High level view of the tool</td>
<td>86</td>
</tr>
<tr>
<td>Figure 46:</td>
<td>Shopping Dossier, high level architecture</td>
<td>87</td>
</tr>
<tr>
<td>Figure 47:</td>
<td>BPMN Hello World diagram</td>
<td>89</td>
</tr>
<tr>
<td>Figure 48:</td>
<td>BPEL Hello World diagram</td>
<td>89</td>
</tr>
<tr>
<td>Figure 49:</td>
<td>A test scenario is represented as a BPEL process</td>
<td>90</td>
</tr>
<tr>
<td>Figure 50:</td>
<td>BPEL processes as scenarios templates</td>
<td>91</td>
</tr>
<tr>
<td>Figure 51:</td>
<td>BPEL engine general operating overview</td>
<td>92</td>
</tr>
<tr>
<td>Figure 52:</td>
<td>Original high-level architecture of the testing ecosystem</td>
<td>97</td>
</tr>
<tr>
<td>Figure 53:</td>
<td>Partial view of the exported results in XML</td>
<td>98</td>
</tr>
<tr>
<td>Figure 54:</td>
<td>New architecture with additional components</td>
<td>99</td>
</tr>
<tr>
<td>Figure 55:</td>
<td>SoapUI Extension architecture</td>
<td>100</td>
</tr>
<tr>
<td>Figure 56:</td>
<td>SOAP response from the getAllProjects operation</td>
<td>102</td>
</tr>
</tbody>
</table>
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AXIOM</td>
<td>AXIs Object Model</td>
</tr>
<tr>
<td>BPD</td>
<td>Business Process Diagram</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modeling Notation</td>
</tr>
<tr>
<td>BVA</td>
<td>Boundary Value Analysis</td>
</tr>
<tr>
<td>CBD</td>
<td>Component Based Design</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>CR</td>
<td>Change Request</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>DPE</td>
<td>Dead Path Elimination</td>
</tr>
<tr>
<td>EFSM</td>
<td>Extended Finite State Machine</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecom Standards Institute</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FSP</td>
<td>Finite State Process</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LoC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MBT</td>
<td>Model Based Testing</td>
</tr>
<tr>
<td>MEP</td>
<td>Message Exchange Pattern</td>
</tr>
<tr>
<td>MTOM</td>
<td>Message Transmission Optimization Mechanism</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>Promela</td>
<td>Process Meta Language</td>
</tr>
<tr>
<td>PTR</td>
<td>Problem Tracking Record</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SD</td>
<td>Shopping Dossier</td>
</tr>
<tr>
<td>SDAT</td>
<td>Shopping Dossier Automated Tester</td>
</tr>
<tr>
<td>StAX</td>
<td>Streaming API for XML</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery Integration</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WS-*</td>
<td>Web Service Specifications Stack</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Definition Language</td>
</tr>
<tr>
<td>WSFL</td>
<td>Web Services Flow Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>
Introduction

Since the advent of the Internet, and more recently of the Web in 1993, software tends to blend naturally with the network. Internet applications are now widely spread, and one can easily deduce that Internet applications seemed to be the wave of the future as stated by Myers in (1). Today, a company must have a presence on the Web, just for simple advertisement purposes or for offering services to its customers. Internet applications are essentially client-server applications, where the client is a Web browser and the server is a Web or an application server. The goal of testing such application is not different from traditional applications. But in the case of Internet application, great care should be put on finding errors before you deploy your application on the Internet, because millions of users may be affected if something is wrong with your application, and you can potentially loose lots of future customers. People have high expectations concerning Web applications because of the intensive competition among companies on-line. If your application is flawed, the user will go and use your competitor’s Web application in few clicks.

Like it or not, a Web application is now becoming the first impression a customer gets from the company, so you better have your Web application up and running correctly. A Web application can generally be divided in two: the client side and the server-side. The client side is a graphical user interface with which customers interact with the application, and the server-side part is where the business logic resides and is processes. Both parts are important and cannot really exist without one another, but making sure that the server-side components do the job correctly is really important.

The new generation of server-side component technology, Web Services, has been widely adopted by enterprise companies, which use them massively not only to provide standards-based services to their customers, but also in their internal infrastructure so to integrate all the different components the development teams develop. But what was true about software testing 20 years ago is still true today, that is, it still remains among ‘dark arts’ of software development according to (1). Add the fact that Web Services is a relatively new technology; sound and complete methodologies for testing Web Services are still emerging. The academy is doing a lot of research on this particular topic, and several companies focus their attention on developing tools for testing Web Services.

Web Services testing is a topic that was, is and will continue to be thoroughly addressed by the software industry.

1.1. Problem statement

This thesis is partially based on an internship done at Amadeus IT Group, where I joined the Shopping Dossier team in order to help them test their components. The Shopping Dossier is a middleware framework which groups together several business components such as Booking Logic, a component with which you can book flight tickets, or Fee Logic which is responsible for calculating the fees for the car, hotel, and flight ticket you would like to buy. This framework is mostly used in travel agencies, where travel agent operators, through a Web-based graphical interface interact with the various components that the Shopping Dossier offers. Prior to go into production, the framework has to pass various test scenarios which are handed to the team by another one in an informal textual format. These scenarios, which simulate real use cases, are quite difficult to create and execute due to the complex nature of the framework which gives access to several underlying components. Starting with the fact that the Shopping Dossier components will be exposed as Web Services, this thesis will address the Web Services testing problem in more general. The team expects a tool which will help it creating and executing powerful tests as easy as possible, by automating tedious parts of the process.
1.2. Motivation

Finding motivation for testing software may seem obvious. Because of the massive presence of software in our world, checking their (correct) behavior is essential. Testing accounts for most of the project budget and time, there is a need to lower the costs without jeopardizing the software quality.

The motivation of this project is to ease the creation and the execution of test scenarios which use the different components of the Shopping Dossier. The different components of the Shopping Dossier were about to evolve towards a Web Services-based architecture, so it was decided that I will target the new architecture. Such tool has to cope with the various standards used by Web Services, and should provide mechanisms to tests the different functionalities a Web Service offers to a service consumer.

1.3. Goals and expected results

The goals of this thesis are the following:

1. Background studies on:
   a. Software testing, by introducing several concepts of this software engineering area.
   b. Server-side components, by presenting some popular technologies such as CORBA, RMI and Web Services.

   a. A prototyped architecture made of several components, based on a solution which was proposed while at Amadeus will be detailed.
   b. Some components prototypes which enhance the ecosystem will be presented and their implementation as well as their testing will be discussed.

The expected results are mainly:

- A compact introduction to software testing and its various types.
- A brief overview of popular technologies for building server-side components, as well as a more detailed overview of the Web Services technologies.
- An investigation on methods for testing Web Services and composite Web Services.
- A testing ecosystem which will answer the Shopping Dossier team’s needs.

The test scenarios are modeled using BPEL, and therefore the ecosystem involves various tools which we need to create and execute BPEL processes:

- A tool for designing BPEL processes (Eclipse BPEL Designer)
- A tool for deploying and executing the processes (Apache ODE)
- A tool for testing Web Services (SoapUI)
A SoapUI Extension is also proposed and discussed in detail. This extension enables a user to remotely execute tests from a Web application. As a proof of concept, a small Web application was developed which uses this extension. Finally, another small Web application was developed, which aims at monitoring deployed BPEL processes, in order to check out the current status, and the internal variables of the processes.

### 1.4. Thesis structure and outlines

The first part of this thesis is more theoretical and introduces the various parts which make up this thesis’ title, that is, software testing, test automation, server side technologies as well as a focus on testing Web Services. By taking a look at Figure 1, you will easily understand the direction the thesis will take and some of the questions it tries to answer.

The second part of the thesis is more practical and will talk about the approach I came up with for testing Amadeus’ components during my internship, as well as some details about the architecture and implementations of prototypes I have developed as proof of concepts with the goal of automating the Web Services testing process.

![Figure 1: How to understand the thesis' title](image-url)
Chapter 1

Chapter 1 gives an overview of the consequences of software errors, and introduces software testing and its various types. It also talks about methods for measuring quality and some challenges in software testing.

Chapter 2

Chapter 2 introduces software components, with a little history background and also presents the evolution of server side technologies. An emphasis will put on Web Services which are extensively used to implement the business logic of server-side applications.

Chapter 3

Chapter 3 briefly introduces the standard Web Service composition specification: WS-BPEL 2.0.

Chapter 4

Chapter 4 presents the goal of test automation, manual and automated testing. It argues that test automation is not a universal remedy and reasons on the usage of manual and automated tests.

Chapter 5

This chapter will present the particularity of testing Web Services at two levels: unit and integration. Methods for each level will be given and discussed along with the current efforts in this area.

Chapter 6

Chapter 6 introduces the idea behind the work done at Amadeus IT Group which serves as a case study. It states the problem description, the background, the requirements and expectations. It also talks about the issues that were faced.

Chapter 7

Chapter 7 will investigate the proposed solution, discuss and investigate the choices that were made, and finally present the different parts involved in the ecosystem.

Chapter 8

In this chapter I will introduce and detail the original architecture of the tool as well as a new version which includes new components I developed. All the various components will be discussed, and their architecture and implementation will be detailed.

Chapter 9

Finally, chapter 9 will evaluate the work done and draws some conclusions. Some future improvements will also be provided.
Part 1
Chapter 1

1. Introduction to software testing

“Testing is the infinite process of comparing the invisible to the ambiguous so as to avoid the unthinkable happening to the anonymous”

- James Bach

1.1. Software failures

According to (2), bugs in final products cost the United States economy between $22.2 and $59.5 billion per year. No matter how thoroughly software was built, it contains defects. Those defects can be detected and fixed during the different software development lifecycle’s phases. Testing accounts for a large part in the software development lifecycle, and most of the time it is the largest phase in terms of time and money.

Testing is therefore a very important task that should be taken into great consideration. You want to know why? In order to avoid the following disastrous scenarios:

- The Therac-25 radiation poisoning

The Therac-25 treatment system is a computerized radiation therapy machine which was used to provide radiation to specific parts of the body in order to eliminate malignant tumors in cancer treatment. Therac-25 machines were put in place in 1976 all over North America. The Therac-25 gave to at least 5 people radiation overdoses, resulting in fatalities. The first incident occurred in 1985.

The cause: uncaught errors;

Brief explanation:

The Therac-25 is a software-controlled machine used in cancer treatment which became unfortunately famous for giving to some people radiation overdoses which kill them instead of curing them. Some of the errors were not reproducible and therefore it was hard to fix the defects.

- The system administered a very large overdose to a 61 year old woman, essentially burning her. The hospital and the technician denied any misuses, so she kept receiving treatments. She had to have her breasts removed and she lost the total usage of one arm.

- A second incident occurred to a 40 year old woman who was receiving her 24th Therac-25 treatment. The machine kept stopping 5 seconds during the treatment with an error. The technician, seeing a ‘No Dose’ message appeared on his screen kept pushing the ‘P’ key to proceed with the dose. This was done 5 times, giving the patient a dose of 13,000 to 17,000 rads. She died 3 months after.

- Other cases can be found in (3).

1 A rad is an obsolete unit of absorbed radiation now replaced with the gray. A normal cancer treatment is around 200 rads. 1000 rads of radiation to the entire body is considered to be fatal.
- Lufthansa A320-200 plane crash

The Lufthansa A320-200 was an Airbus plane which departed from Frankfurt bound to Warsaw, where it crashed on 14th September 1993, killing one crew member, one passenger and injuring 56 other persons.

The cause: one critical condition was not met;

**Brief explanation:**

The Lufthansa Airbus A320-200’s plane failed to stop correctly during its landing and crashed due to failed thrust reversers’

1. Landing gear must be down
2. The wheels must be turning
3. The weight of the plane must be carried on the wheels.

Conditions 1 and 2 were met, but because of the vast amount of water on the runway, the wheels suffered from a hydroplaning effect and an unexpected tailwind did not provide enough lift on the wheels, so the control’s software was fooled into thinking that the third condition was not met. The thrust reversers only deployed 9 seconds after landing, thus not providing enough power to decelerate on time. The plane overshot the end of the runway, causing two fatalities and injuring 56 people. More precise details can be found in (4).

More notorious software failures can be found in Appendix A.

As you probably understood, the best case scenarios of software flaws range from annoyed customers and/or scientists, money losses (which can be rather substantial in some cases), to human’s lives losses as in the Therac-25 and the Lufthansa A320-200 cases.

Due to the increased complexity of software systems, testing is really hard and engineers, even the brightest ones, cannot come up easily with all possible exception scenarios that will lead to a failure. Even if all the underlying components which form a system were meticulously tested, the combination of components and their interactions can bring unexpected and potentially erroneous behaviors difficult to discover. When complex systems fail, the scenario which led to that particular malfunction can rarely be imagined by a sane human being in advance, and even though s/he came up with it, millions of other cases may be overlooked.

To quote the legendary computer scientist Edgser Dijkstra in (5):

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence

---

2 Thrust reversal is the effect of changing the propeller’s pitch so to direct the thrust forward in order to slow down the plane. It is applied immediately after touchdown, along with spoilers’ deployment, so to increase deceleration.
1.2. Software testing

Today more than any other day, software’s are massively part of our everyday’s life. Just think at the ATM when withdrawing money, placing a call to a friend with a new smart phone, or even the sweet voice in our car guiding us on highways. Therefore, since we are using them on a daily basis, we must blindly “trust” them, because if we spend our time doubting their faithful behavior, you will end up staying at home not moving at all.

So, how can we trust software to do **correctly** what they were designed to do? In the early days of software engineering, developers knew that they couldn’t deliver their special product without making sure it is working, with a specific approach. They **tested** their software just like a car manufacturer does **crash tests** to make sure the car is somehow reliable in case of a collision. How many of you would buy a car nowadays without taking a look at the numbers of star(s) the European New Car Assessment Programme (Euro NCAP) attributed to the car? If you are a responsible driver, you will look at those bright yellow stars just to make sure your money is put into a car that does worth it and that will protect your beloved ones in case of an accident.

The same approach can be used for purchasing software’s, but as you know, it is quite different from a car, because of its non-physical properties. Let us put the piece one by one in order to set up the context:

According to (6), ‘testing is an activity performed for evaluating a product quality, and for improving it by identifying defects and problems’.

This definition can hold for every product, but must be refined a little for the software area because of its differences with other engineering fields.

A polished description according to (6) defines it as following:

‘**Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior**’.

In the above definition, every word is important, but some of them need to be further detailed as they are in (6).

- **Dynamic**: the term enhances the fact that software is a running piece of code with data that is fed to it. Depending on this data, the system might behave differently, in a non-deterministic way. This specific data is termed as the *input* of the system.

- **Finite**: because of the structure of a program, executing all possible combinations is sometimes infeasible because it would take months or even years. Different techniques exist to cope with programs suffering from the *state explosion* problem.

- **Selected**: When testing software, the tester expert has to choose the most relevant tests in order to maximize the quality of the software. Such choice is challenging and needs domain expertise. The tester has to find the right balance between the level of confidence that the tests being executed gives back, and the budget and time constraints.

- **Expected**: Obviously, when you want to test your system, you must check its current behavior against a good behavior you identified as acceptable.
Two important distinctions have to be made that surround the area of software testing and might confuse people, that is, validation and verification (V&V).

**Validation**

Validation is asking ourselves the following question: *are we building the right product?* This phase generally includes the end-users of the product to ensure that the software meets their expectations.

**Verification**

Verification is asking ourselves another question: *are we building the product right?* In this phase, we generally check if the system meets the specifications.

Testing, validation and verification are all complementary notions and activities that aim at improving the overall quality of the software by preventing defects to occur by making sure that what we are actually developing is the right software, and that we are making it correctly.

Testing is no more an activity which is done after the code is written, but rather an activity that should cover the entire development cycle.
1.3. What type of software testing does exist?

Because testing is an activity that can be found throughout the software development cycle, various levels of testing emerged for different scopes.

1.3.1. Objectives of testing

Prior to introducing the most important levels of testing, one has to set the objective of testing. A test is conducted with a particular goal in mind. This goal has to be precisely defined in order to quantify it (if possible) throughout the test process.

When checking if the system is correctly implemented, that is, that its functionalities are correct, we generally talk about ‘functional testing’ as opposed to ‘non-functional testing’ which aims at checking other properties of the system such as security, performance and so on.

1.3.2. Unit testing

Unit testing defines the verification of a piece of software in isolation, that is, without looking into its dependencies. In (6) it is defined as following: “unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be individual sub-programs or a large component made of tightly related unit”.

Why does it depend on the context? Because for some programmers, unit testing can be the smallest piece of structured logic within a source code, for instance a method, whereas for larger system, a single component in a larger system can be considered as a unit.

The context is really important because the definition itself of unit testing is subject to interpretation, as discussed in (7).

1.3.3. Integration testing

Integration testing is defined by (6) as the ‘process of verifying the interaction between software components’. Testing individual piece of code or component to check if they behave good on their own is good, but one cannot ensure a proper behavior if you do not test interactions between the components which forms your system.

Some companies fully delay the integration phase after every unit was methodically tested, and failed to catch maybe important defects earlier in the development cycle. Because of that, the expression ‘integration hell’ is coined because people perfectly know that the list of bugs is about to get longer during the integration phase.

An approach to overcome this concern is to continuously and incrementally integrate from the beginning of the development cycle. This is known as ‘continuous integration’ or CI, and it is described by Martin Fowler in (8) as the ‘practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.’

---

3 Also known as correctness testing or conformance testing.
4 Software components are discussed in more detail in the next chapter.
5 Also known as the ‘Big Bang’ approach as stated in (83).
1.3.4. System testing

System testing’s purpose is to check the overall behavior of the whole system in general as stated in (6). If done correctly, unit testing and integration testing should have caught most of the bugs. Therefore, system testing is more concerned with non-functional aspects of a system, such as security, performance… The deployment environment, different involved hardware is also checked at this level.

All of the above main testing levels generally fall in two broader categories, that is, white box or black box testing.

White box testing\(^6\) simply means we have the knowledge of the internal structure of the system we are testing, i.e. we have access to the source code or to a detailed behavioral model of the system.

Black box testing on the other hand assumes that the tester does not have any clue of the internal structure of the software. Therefore we are limited to its visible interfaces, where we generally find input and output properties of the system under test.

Testing with the knowledge of the internal structure is obviously not the same as when we don’t, and the techniques used to test a system are fairly different depending on the perspective we choose to take.

1.3.5. Other types of testing

Many other types of testing are also important, as mentioned in (6), such as:

- Usability testing
  - Evaluates the software in term of usability, if the documentation is correct, how the software answers the needs of users…

- Acceptance testing
  - Checks the behavior of the system against customer’s requirements.

- Installation testing
  - Verifies the software after installation on its deployment environment. Some specifics features of the installation can also be reviewed.

- Alpha and beta testing
  - Very popular, this kind of testing involves not fully completed version of the software that are made available to the end-users (generally for free), so they can give relevant feedback on the system’s feature. Alpha generally means that the software is tested within the company, whereas beta is for the public. But it seems that the trend is to also let people get their hands on alpha versions so to give feedback earlier in the cycle.

- Reliability testing
  - In order to improve the reliability of the software, this phase aims at testing the robustness of the system by checking how gracefully it degrades in presence of bad inputs, a network partition…

\(^6\) Also known as glass-box testing or structural testing
- Regression testing
  
  o Defined by (6) as the ‘selective retesting of a system or component to verify that modifications have not caused unintended effects…’

  This kind of testing can lead to the so called ‘minefield paradox’\(^7\). Let me explain this analogy first presented by Brian Marick and also mentioned and illustrated in (9).

- Performance testing
  
  o This kind of non-functional test aims at testing the performances of the system. It can be a response time, memory usage…

- Stress testing
  
  o This testing technique evaluates the system at the maximum load, and even beyond, it can handle. For instance, you test if your Web application can handle its fixed limit of 10,000 users at the same time you test for more than this value.

- Configuration testing
  
  o If your system was designed to cope with different levels of users, you might test for various configurations of the system and inspect its behavior.

\(^7\) Best known as the ‘Pesticide Paradox’ coined by Boris Beizer in (13).
1.4. Measuring the quality

The goal of testing is to reveal defects, and therefore selecting tests that have a good probability to detect errors is critical. As we know, complete testing is impossible due to the fact that inputs can take infinitively different number of values, and yet, we have to try to reach complete testing with the time and the budget we are given. In software testing, we generally rely on metrics to measure how well tested a SUT is. Most of the metrics that will be presented were presented and discussed in (1) and in (10).

1.4.1. Black box testing methods

In black box testing, we can generally rely on the following methods to find defects in the SUT:

- Equivalence partitioning
- Boundary value analysis
- Cause-effect graphing
- Error guessing

**Equivalence partitioning**

Equivalence partitioning is a technique which divides the input and output domain of a system into classes of data, called *equivalence classes*, from which test cases can be derived. Within an equivalence class, values are said to be equivalent when certain inputs yield the same expected result.

For example, one method receives a ‘day’ of a date as an integer parameter. In a programming language, a day value ranges from 1 to 31 (or 0 to 30). Therefore, a *valid day* could be one of the 31 choices, and an *invalid day* would be a value outside that range, -1 or 32. Those ranges are called *partitions*\(^8\). This concept is used also in the UML Testing Profile (11), in the form of a DataPool. One can create a partition containing correct value which will yield good expected results, as well as a partition which will contain data that is likely to provoke errors in the SUT. In testing literature, this kind of practice can be found under a larger umbrella called *Boundary Testing*.

---

\(^8\) Also known as *category partition*
Boundary value analysis

Boundary value analysis (BVA) is a data selection technique where extreme values of data are chosen. The values generally include maximum, minimum, inside and outside boundaries, particular data values and error values. This can be seen as an extension of the equivalence partitioning technique and complements the latter perfectly. For instance, when testing a condition such as:

- if (x>0 and x<32), where x represents a day of the month.

You can design test cases with the following data values:

- The maximum: 31
- The minimum: 1
- Maximum +1: 32
- Minimum -1: 0
- An error value that may enforce an exception mechanism to be activated: e³
- A value inside the range: 15

This technique can help you create robust test cases that emphasize on error handling, which will help you discover errors in behaviors. However, this technique does not apply when facing unbounded data value or when using weak-typed languages where the type of the data is determined at runtime, therefore not knowing in advance the real type of the data and how it will be interpreted.

Error guessing

Error guessing is a process of making an educated guess on where errors can be hidden. This technique requires a good domain expertise, and can rely on past experiences with other techniques such as the ones previously mentioned. You can also base your tests on parts of the functional specifications that has not been addressed or when you find out an error which was not classified nor documented.

Metrics used in BVA can serve as a basis for error guesses in error-prone situations such as:

- Data initialization; to see if the data was properly instantiated
- Real data handling; to check out if the data involved is real and not data which programmers tend to use so it will reflect best expected results
- Proper error handling; to check if errors are prioritized when facing with multiple errors, or also to see if error messages are clear enough

Opaque specifications can also be the basis for educated guesses.

Example of an unclear specification:

"An isosceles triangle area computation function will receive 3 values, where 2 of them are equal, and the function will return the area"

This kind of specification does not say anything about errors handling when not receiving 2 identical values. So one can create a test that will check what happens when dealing with different values.
**Cause-effect graphing**

Cause-effect graphing is a technique which uses directed graph to explicitly map a set of causes to a set of effects. This technique can explore combinations of inputs yielding a certain result, which is not possible with BVA or with equivalence partitioning. They provide a concise illustration of logical conditions and consequent actions.

Certain cause-effect graphs may also include constraints to further enhance the representation of certain effects. Finally, cause effect graphs can be converted into decision matrixes, which serve as a basis for creating test cases. More details are given in (1) about the cause effect graphs technique which can be rather useful to discover ambiguities in a system’s behavior.

**Figure 2: Basic cause-effect graphs**

*In a) a certain cause C1 has a value which will result in an effect E1, and a function identity will state this fact. In b) the graph has an intermediate node I2 which proceeds an effect. You can also make use of certain logical functions such as **and** in c), **or** in d) and a negation function as in e)*

C1 → identity → E1  
C2 → I2 → E2  
C3 → C4 → E3  
C6 → E4  
C7 → E4  
C8 → E5

In a) a certain cause C1 has a value which will result in an effect E1, and a function identity will state this fact. In b) the graph has an intermediate node I2 which proceeds an effect. You can also make use of certain logical functions such as **and** in c), **or** in d) and a negation function as in e)
1.4.2. White box testing metrics

In white box testing, the metrics are much more complex. First, the metrics can be divided into three main categories, depending on what you want to test:

- Control flow
- Data flow
- Transitions

Within each of these ones, you can find several sub-metrics as you can see in the following picture, redrawn from (10).

![Figure 3: White box testing related metrics](image-url)
I will briefly explain some of the sub metrics which are defined in (10), using a small example created by the author in (1):

- **Control flow**

  When using this metric, a test should execute every reachable statement of a program at least once. For instance, in the above example, we would have to ensure to create test with specific values for \( a \), \( b \) and \( x \) so to go through each assignments at least once.

- **Decision coverage**

  Decision coverage can be seen as a stronger version of statement coverage in the sense that you have to create tests for conditions that have true outcomes, and test with conditions which have false outcomes. Decision coverage usually requires satisfying statement coverage. In the above example, you would have to create test for the \( if \) conditions that will generate a true, with \( a=3 \) and \( b=0 \) and \( x=2 \), and with values not satisfying the conditions to pass through the \( else \) part of the branch (not present in the example). Note that decision coverage focuses on the outcome of the whole expression, that is in the example in the outcome of \( if (a>1 \: && \: b==0) \) and not the outcome of sub-expressions such as \( a>1 \).

- **Condition coverage**

  A test achieves condition coverage when it evaluates all conditions of an expression to true for some tests, and false for other one. In the example, you should have test with a value satisfying \( a>1 \) and another test which doesn’t, and that for every condition independently from each other. It is more or less the same as decision coverage, but with more sensitivity to the control flow.

- **Path coverage**

  [Known as branch coverage]

---

Figure 4: Small program to be tested, visual representation and Java code (courtesy of G. Myers, redrawn)
Test should execute possibly every path through the control flow graph of the program as illustrated in Figure 4. As you can see, there are 5 paths: a, b, c, d, e. For this example, it may seem straightforward, for instance test may execute path \textit{abd} or \textit{ace}. But the presence of loops gives an unbounded number of possible paths. Even if statements can generate large number of paths. For instance, suppose a method with (only) 10 \textit{if}-statements, you will end up having $2^{10}$ possible paths and therefore the same number of tests. Adding only one more \textit{if}-statement will double it up resulting in $2^{11}$ tests to create. This issue is referred to as the \textit{state space explosion problem} in the model checking area or \textit{combinatorial explosion} in mathematics.

- Multiple condition coverage

A test set achieves multiple condition coverage when all possible combination of condition in each decision has been satisfied. Imagine a decision with 2 conditions such as \textit{if (a>1 \&\& b==0)}. It will require $2^N$ tests where $N$ is the number of conditions, that is, in our little example 4 tests. This is generally feasible for decision with a small number of conditions.

- Decision/condition coverage

When a test set achieves both decision coverage and condition coverage.

\textit{Data flow}

Data flow graph is a special control flow graph with added information which focuses on the values of data along a program. This added information concerns the \textit{definition} and the \textit{use} of a data variable, where a \textit{definition} can be considered as a \textit{write} of the variable and the \textit{use} as a \textit{read} of the variable. More precisely, a \textit{definition} is a statement which sets the value of the variable, and the \textit{use} is any expression or predicate which make use of that \textit{definition}.

```java
public void bar(int a, int b, int x) {
    if (a > 1 && b == 0) {
        x = x / a; \textit{A definition of variable X}
    }
    if (a == 2 || x > 1) {
        x += 1; \textit{A use of variable X}
    }
}
```

\textit{Figure 5: Definition and use in data flow testing}
A def-use pair for variable \( v \) is a definition of \( v \), a use of \( v \) and a \( v \) definition-free path from a definition to a use of variable \( v \). In other words, the path ensures that variable \( v \) is not overwritten by a statement before it is read by an expression or used within a predicate. In Figure 5, you can see a def-use pair for variable \( x \), with no statement modifying the value of \( x \) between the definition of \( x \), where it takes a value, and the evaluation in the if-statement.

According to authors in (10), a def-use pair is feasible if there exist some test inputs can cause a definition of a variable and then a use of the same variable to be executed, otherwise it is infeasible.

- All-uses

This criterion ensures that a test set will check all feasible uses for all definitions of a variable.

- All-defs

This criterion ensures that a test set will check at least one def-use pair for all definitions of a variable, that is, at least one path from each definition to a feasible use.

- All-def-use-paths

This criterion ensures that a test set to check out all the paths from a definition to a use of a variable. This is generally infeasible in practice because it would require a large number of test cases.

**Transitions**

When systems become too complex, there is a need to use abstraction methods, i.e. a need to reduce the complexity of a whole system, or just a part of it, using models. When using models, one can focus on a certain behavior of the system s/he would like to test by reducing some irrelevant details of a system. These abstractions can make advantage of widely accepted notations such as:

- Finite State Machine (FSM)
- Extended FSM (EFSM)
- Labelled Transition System (LTS)
- UML State Machine diagram
- Finite State Process (FSP)

\(^{10}\) Known also as *Finite State Automaton*
The different notations have commons concepts such as:

- **State**
  A state is a unique *configuration* of information in a program.

- **Transition**
  A transition is a change from one state to another state.

- **Actions**
  An action is a description of an activity to be performed. Actions can be of one of the following type:
  
  - Entry action: activity performed when entering a state
  - Exit action: activity to perform when exiting a state
  - Input action: activity to perform which depends on the current state and input conditions
  - Transition action: activity to perform when performing a certain transition

You can see a simple example of a FSM diagram in Figure 6, which represents the two different states a door may be in, the transitions from one state to the other and the entry actions to enable the state. You can also visualize a different FSM diagram in Figure 7 with additional states, which will be used to describe some coverage criteria.
Coverage criteria using these notations is equivalent to a search of reachable states, that is, trying to find paths that will reach the state we want to be in. In general, you can find the following coverage criteria:

- All-states
  Every state in the model must be visited at least once during the state-space search. In the above example, you can write a test which go to A;C;D;F to cover all states of the model.

- All-transitions
  Every transition must be traversed at least once. This can be done with two test, traversing A;C;D;F and B;E transitions.

- All-transitions-pairs
  Every pair of adjacent transitions should be traversed at least once. For instance, for State S3, adjacent pairs are: B;E, B;D, C;E, C;D. The number of tests per state is generally the product of the indegree and outdegree of the state node, 4 in our example for S3.

- All-paths
  Every path should be traversed at least once. In Figure 7 it will require the creation of tests which traverse A;C;E, A;C;D;F, B;E and B;D;F. In the presence of loops, that is in presence of self-transition in the FSM model, it will result in an infinite number of path.

- All-loop-free-paths
  Every loop-free path must be traversed at least once. In our example, the tests would be the same as the one generated for all-paths coverage because our FSM model is loop-free.
1.4.3. Which metric to choose?

Well, it really depends on the kind of system you want to test, if your system is more data-driven or if it contains lots of control flow logic, and if you use higher level notation using state-based model, it will depend on the type of model you extract from your real code. But what can be said without any hesitation is that a combination of coverage criteria will result in increased chances to find defects in your SUT.

The list of metrics is obviously not complete, but it shows some important coverage criteria you might use when creating tests. Further coverage criteria can be found in (12),(1) and(10).

Concerning the high level notations such as FSM, EFSM… they all have a strong mathematical background, which make them fairly precise, semantically speaking, and their use is a core concept of a new form of testing: Model-based testing (MBT), but they play also an important role in Model Checking.

Briefly speaking, model checking is the process of verifying that a given property (i.e. a predicate) is satisfied in a model. A model can be any formal representation of a system, using FSM for instance, and a property can be expressed in any logical language such as temporal logic. A model checker will search the state-space for a state in which the given property is satisfied, and if so will return ‘ok’, if not, will return an error. Some model checkers will even return the path that leads to the state presenting the error. In theory, a model checker can exhaustively search the state-space, in order to prove the correctness of the system and/or to find bugs.

MBT, in few words, is a technique which helps you deriving test cases from the model of your program. One major drawback of using state-based models, is when there is a large number of states, the state-space search becomes infeasible. Techniques to reduce the size of the state-space do exist such as partial-order reduction as presented in (13).
1.5. Challenges in today’s software testing

Even though testing is a well accepted discipline among the industry, it still remains, in practice, an ad hoc, expensive and unpredictably effective approach. In (14), the author points out the huge gap between researches and the industry’s difficulty to practically apply the theory and methodologies.

New software computing paradigm needs all the expertise gained by software engineers throughout the past five decades, but need to be constantly adjusted in order to ensure a high level quality of service. Reducing the costs of testing without jeopardizing the quality is one of the main drivers for test automation in the industry.

Distributed systems, because of their nature, are more difficult to test. Concurrency between components, race conditions, absence of a global clock…are some of the causes of the rising complexity inherently present in distributed systems.

In order to keep up with the quality of software, software engineering researches keep focusing on full test automation as Antonia Bertolino pointed out in (15). She argues that the quality analysis and testing have to keep the pace with the growing quantity and complexity of software. This is one of the (many) reasons why test automation was, is and will continue to be a critical research area. Even though she outlines the great efforts made in the recent years with the numerous framework of the xUnit family available, she argues that in order to push the automation a little further, we must enhance test frameworks with test generation and environment simulation capabilities. She noticed the trend to combine different techniques such as static code analysis\(^\text{11}\), symbolic execution\(^\text{12}\), formal verification\(^\text{13}\)…and she said that “it could be the really winning direction towards the 100% automation dream”.

Another big challenge in software testing, aside from automation, is people. Testing is sometimes seen as boring or as a necessary evil in enterprises. And no matter what you think, convincing management that resources and money should be devoted to testing is still the key to a successful and adequate testing process. Communication between testers, developers, managers and end users will greatly benefit to everyone in understanding what the client really wants, and in building what is needed, no more no less.

SOA, in the form of Web Services, has a huge impact on the IT landscape bringing new challenges for software testing. For instance, the proper location of the software can be outside user’s control, and because of that, changes into the service’s implementation are completely unpredictable to users. Keeping functional test in sync with the service’s functionalities is critical, requiring a strong but flexible test framework to work with.

Testing a service-based application bring this new kind of question: who is responsible for its testing? The service provider, the service requester or both? Not to mention that trust is a central issue over the Web, so security testing is also very important.

\(^\text{11}\) Static code analysis is the fact of analyzing the code without executing it.
\(^\text{12}\) Instead of using normal test inputs, one supplies symbols representing arbitrary values.
\(^\text{13}\) Formal verification is the act of proving (or disproving) that the software is correct with respect to certain formal specifications or properties.
Chapter 2

2. Server side components

“Divide et impera”
(latin quote for ‘Divide and conquer’)

2.1. Software components

Let me give you a definition of ‘Component based design’, known as CBD as stated in (6):

“A software component is an independent unit, having well defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse.”

- Independent unit
  - Meaning that it should be a self-sufficient element that does a specific unit of work. Generally, a component provides a common roof for its function and data.

![Figure 8: An ‘Authorization’ component which can be used for security purposes.](image)

- Well defined interfaces and dependencies
  - The way that the component is presented to the external world should be clear and possibly unique, and any external elements that this component needs in order to fulfill its computation have to be specified.

![Figure 9: In red (login, register) you can see the interfaces that the ‘Authorization’ component provides, and in blue the dependency it needs (CheckIfUserExist).](image)

- Can be composed
  - It means that this particular component can coexist with other components in order to create a specific set of functionalities.
  - In the previous figure, you can see that the ‘Authorization’ component is actually part of a bigger system including a ‘Database’ and ‘Web Application’ components that work together to provide a service to its users.
- Deployed independently
  
  o Deployed means that the component will exist within a particular environment. This environment should provide all the necessary plugs in order for the deployed component to work properly.

CBD is just the natural result of what human minds tend to do when facing complexity, dividing the problem into smaller ones, and building relative solutions to each of them, and eventually glue them (read *re-compose them*) to form a coherent solution.

2.1.1. History and goals of software componentization

Componentization of software is not a brand new idea, in fact to be accurate, it started back in 1968 during a NATO conference in Germany on software engineering with Dr Douglas Mellroy’s paper “Mass produced software components” published in (16). It was a response to the so called ‘software crisis’, term used in the beginning of the software engineering area describing the rapid increase of computational power and the complexity that will arise from the fact that it will harder and harder to write correct, understandable and verifiable computer programs.

To quote Edgster Dijkstra in (5), “more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in the state of eternal bliss of all programming problems solved, we found ourselves up to our necks in the software crisis!”

Shortly said, Dijkstra held responsible the fact that because computers are more powerful, i.e. power of available machine grew by a factor more than a thousand; society’s ambitions to apply those machines grew in proportion. And who was caught in the middle of this tension between the ends and the means: the poor programmer. According to him, it is in that same NATO conference that the world acknowledged that the software industry had to change the way to develop software.

That being said, components as basic blocks for software development bring many advantages. Generally they follow the axioms of the object oriented development; flexibility, extensibility and reusability. Being able to manage the complexity in large application, improving the overall quality by integrating well-tested part, managing the only true constant in computer science, i.e. the ‘change’, by modifying only parts of the systems are the benefits of component-based software development.
2.2. Server-side components

Before diving into server-side components, let’s briefly show the evolution of components throughout the history of computing, where we started from simple objects to evolve into a newly accepted form of distributed computing: Web Services.

Figure 10: Components’ history evolution. (Diagram based on lectures notes of Martin Henkel (lecturer at KTH) in Model Driven Architecture of Components course)

- **Object oriented programming**

  Examples of OO languages: Java, C#, Simula 67…

  The applications that were built in the 60’s until the early 90’s were generally used within a single address space, that is, on a single machine in a closed and homogeneous environment, and most of the time the systems were built in a language dependent fashion.

- **Distributed objects**

  Examples of distributed objects: Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM)… Language independence came along this new architectural model, along with the fact that the environment is LAN-distributed, generally heterogeneous.

- **Components**

  Examples of components models: Enterprise JavaBeans (EJBs), Component Object Model (COM)… Components can be based on distributed objects with some extending features like transactions, security, concurrency…

- **Web services**

  Web services are one of the latest evolutions in distributed computing. With this new architectural model that relies mostly on components, but this time we work in an Internet-distributed environment, i.e. highly heterogeneous, with interoperability in mind, that is, no binary protocols.
Server-side components are, simply put, distributed objects. To be a little more accurate, server-side components are grouped objects, which compute specific tasks, generally used on middle-tier application servers which manage the components at runtime and make them available to remote clients across the network. Server-side components are pieces of software which are remotely called from a client using a specific communication protocol. As you can see in Figure 11, a client calls the createAccount method which resides on a remote node through the network.

![Figure 11: Typical client/server architectural style, where the cloud represents the Internet](image)

There are four major distributed objects paradigms from which you can create server-side components; Microsoft’s DCOM, Object Management Group’s CORBA and Sun’s Java Remote Method Invocation (RMI) and Web Services (WS).

Let’s see briefly what they are made of.

### 2.2.1. CORBA

CORBA is an OMG’s specification describing an architectural style and enabling the development of distributed software components. Because it is a specification, it can freely be written in multiple languages.

CORBA relies on a protocol called the Internet Inter-ORB Protocol (IIOP), which is a special implementation for TCP/IP of a more abstract protocol, the General Inter-ORB Protocol (GIOP), for accessing remote objects. The central feature of CORBA is the Object Request Broker (ORB), which are simply said pieces of middleware that allows making programs calls from one computer to another via the network. The goal of every distributed paradigm is to create the ‘illusion’ that the object that your program is calling resides in the same address space, i.e. that every interactions between the objects appears to be transparent whether the objects are local or remote ones. The ORB provides that kind of transparency for you. The ORB uses an Interface Description Language (IDL) to describe the data which is to be transmitted when doing remote calls.

![Figure 12: A client sending a request to an object through the ORB](image)
The ORB will find the object’s implementation; it will prepare the object to receive the requests made to it, communicating the requests to it and carrying the reply back to the caller. The CORBA object interacts with the ORB through the use of an ORB interface or through an Object Adapter. The interface the client sees is completely independent of the location, the programming language it is implemented in or any other aspect that is not reflected into the object’s interface, thus guaranteeing location transparency and language independence.

### 2.2.2. DCOM

Microsoft’s proprietary distributed object paradigm, called DCOM, relies on a protocol called ‘Object Remote Procedure Call (ORPC)’. DCOM relies on COM objects, which defines a non distributed component model. DCOM is integrated with other Microsoft’s products in order to form a distributed network architecture, that is:

- ActiveX, a specialized component to be used with Microsoft Internet Explorer (IE) (either as IE plugs ins or ActiveX controls)
- Microsoft Transactions Servers (MTS), special server used for dealing with multiple-transactions in distributed applications.
- Microsoft Message Queue Server (MSMQ), a special server generally known as a Message Oriented Middleware (MOM) that ensures the delivery of messages from application to another one.

Just like an ORB object, a COM object has interfaces which can be called by remote clients. A DCOM server acts as a container of objects, which are accessible at runtime to the client.

### 2.2.3. Java RMI

Sun’s Java RMI relies on a protocol called the Java Remote Method Protocol (JRMP). RMI is known to be an object oriented version of Remote Procedure Call (RPC), but the idea remains the same, calling for distant computation. Java relies on Java Object Serialization\footnote{See Marshalling in the next page.}, which allows for data representation to be transmittable as streams. Java Object Serialization is language specific, so both the client and the server objects have to be written in Java. An RMI server object defines, just like an ORB or COM object, interfaces which can be accessed remotely. RMI depends on a naming mechanism called the RMI registry that runs on the server part and contains the information about the available objects. The caller, in order to access objects, has to do a lookup to obtain its reference, and then it can invoke its methods. Java RMI is language dependent, but thanks to the Java Virtual Machine (JVM), you can run applications on different platforms without fearing incompatibilities, because shortly said, when you do distributed applications with Java, its one JVM talking to another one over the network.

Also, for interoperation purposes, a bridge between Java and CORBA was made with the help of a new protocol, RMI-IIOP, which can translate object from one side to the other when transmitted over the network.
The three technologies regroup same core concepts when dealing with remote object communication such as the stub and skeleton.

Machine A wants to use Machine B’s function, let’s say an ‘add’ function. The caller will make a call to its local procedure implemented by the stub; the stub will marshal the call type and data into a request message. The stub will send the message over the network and will block the current execution thread.

The server’s skeleton (which is basically the server’s equivalent for a stub) receiving the request, will unpack the message and look for a procedure reference. When it founds the ‘add’ reference, the stub will unmarshal the procedure’s arguments (in the case of an add function, the arguments are supposedly two integers), then will execute the procedure on the called object. The called object performs its computation and returns the result. The skeleton will pack the returned result into a response message, and send it back to the client over the network.

The client’s stub receives the response from the network, unpacks the output arguments (in our case the added value of the two integers it sent) and passes the output argument to the caller, resuming its execution thread and continuing its flow of actions.

This kind of architectural facility brings an important advantage, that is, neither the caller object nor the called object has to deal with network specific code, because all the functionalities have been moved to the stub and skeleton.

---

15 *Marshalling* (known also as *serialization*) is the process of transforming the memory representation of an object to a data format suitable for transmission over a network.

16 *Unmarshalling* (known as deserialization) is the process of disassembling the message into an equivalent collection of data at the destination.
2.2.4. Web Services

Web Services are an implementation of the Service Oriented Architecture paradigm (SOA), defining a new abstract architectural style which is not bound to any specific technology in particular. SOA architecture defines a paradigm in which a service is the atomic element. A Web service is defined by the World Wide Web Consortium (W3C) as ‘a software system designed to support machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards’.

Web services heavily rely on a lot on open XML-based standards, thus providing seamless integration and interoperability between software applications across the network.

![Service oriented architecture](image)

Figure 14 represents the basic roles played in an SOA infrastructure. As you can see, it consists of three major players: a service provider, a service broker and a service consumer.

**The service provider**

The service provider creates a Web service and publishes its interface and access information to the service broker registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the service... The provider also has to decide in which category the service should be classified in for a given broker service and what sorts of trading partner agreements are required to use the exposed service.
The service broker

The service broker, also known as service registry, is responsible for making the Web service interface and implementation accessible to any potential service requestor. The implementer of the broker decides about the scope of the broker.

Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, the amount of the offered information has to be decided. Some brokers specialize in many listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. There are also brokers that catalog other brokers. The Universal Description Discovery and Integration (UDDI) specification from the Organization for the Advancement of Structured Information Standards (OASIS) defines a way to publish and discover information about Web services. Other service broker technologies include for example ebXML (Electronic Business using eXtensible Markup Language) from OASIS and those based on the ISO/IEC 11179 Metadata Registry (MDR) standard.

The service requestor

The service requestor or Web service client locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.

![Figure 15: Web Services architecture with its core specifications: UDDI, SOAP and WSDL](image)

Web Services are made out of several standards, but five of them are considered to be fundamental. Some existed beforehand and were used to realize Web Services, such as XML and HTTP, and some are specific to Web Services and were crafted for them such as WSDL, SOAP and UDDI.

XML is used as a format the data types, various models and formats. Almost every other standard is an XML-based standard; in fact, Web Services standards are based on XML, XML Schema and XML Namespace.

HTTP is a communication protocols that can be used by Web Services. It is one of the possible protocols that can be used to send messages to Web Services over the network. Other protocols can be used such as SMTP, JMS…
**WSDL**

WSDL, standing for *Web Service Definition Language*, is a W3C standard that describes the interface of a service. WSDL tells you exactly how you can interact with the Web service but says nothing about how the Web Service works. This forces you to rely on only the description of the Web service rather than anything in the implementation - which helps to reduce (maybe even eradicate) compatibility problems between two Web services which provide the same function, but use different implementations. You can make an analogy with IDL and its CORBA object implementation.

WSDL defines two levels of a Web Service, that is, an abstract level and a concrete level.

At an abstract level, WSDL defines a Web Service in terms of the messages it can send and receive, generally using data format based on an XML schema. It defines *operations* that associate a *message exchange pattern* (MEP\(^\text{17}\)), which can differ depending on the concrete messaging protocol being used. It also defines *interfaces* (*"portType" in WSDL 1.1*) which basically groups together operations without any commitment to a transport protocol.

At concrete level, WSDL defines a *binding* which specifies transport and wire format details for one or more interfaces. An *endpoint* (*"port" in WSDL 1.1*) associates a network address with a binding. Finally, a *service* groups together endpoints implementing a common interface.

![Figure 16: WDSL 2.0 general structure](image)

The latest version, at the time of writing, is the 2.0\(^\text{18}\)\(^\text{19}\), but the most used is still the 1.1.

---

\(^\text{17}\) You can read more about the new features for *Message Exchange Patterns* in (19)

\(^\text{18}\) Note: WSDL 2.0 is also a W3C recommendation, whereas the WSDL 1.1 specification is not.

\(^\text{19}\) An example of WSDL 2.0 can be found in Appendix B (B.1).
Types: encloses the data type definitions, generally using the XSD type system. In other words, it is where the data types of the Web Service’s messages are defined.

Message: defines the format of the messages being exchanged between the Web Service and the Web Service’s client. It consists of one or more logical parts. Each part can refer to an XSD element and/or to XSD simple or complex types.

PortType: you can think of it as a container of one or multiple abstract operations, which basically define the signature of a method.

Operation: an operation generally consists of, as in every programming language, input parameter(s), a return value and an optional fault element. A WSDL 1.1 operation supports 4 types of message exchange patterns:

- **Input**: defines the input of the operation, which is a specific message.
- **Output**: defines the return value of the operation. The value is also expressed as a message.
- **Fault**: consider it like an exception message which is returned from the Web Service to the caller in case an error was caught. The error is also contained within a message.

- **Binding**

  It contains the different details of how the abstract elements of a PortType are bound to a concrete set of elements, i.e. it is where you defined which communication protocol the service uses, the data format so on and so forth. The most commonly used communication protocol is HTTP, and SOAP is the most used protocol for the representing the messages between parties. Other data format such as JSON, which is a lighter text-based data representational format, can be used.

- **Service**

  A service is a container which groups port elements together.

  - **Port**

    A port specifies the address, or more precisely the URL, for a binding.

Below, you can have a look at a small example of WSDL 1.1 interface of a *Hello World* Web Service, which offers a single operation *hello*. This method takes a parameter a message containing an xml element of type string and returns a result of the same kind.

---

20 **One-way**: The endpoint (i.e. the Web Service) receives a message.  
**Request-response**: The endpoint receives a message and returns a correlated message to the caller.  
**Solicit-response**: The endpoint sends a message and receives a correlated response.  
**Notification**: The endpoint sends a message.
Types
You can see that there is an inline schema (understand ‘embedded in the WSDL’) which defines two complex elements:

- **hello**: simply defines a sequence of one element, named TestPart, of type string

- **helloResponse**: this element defines the structure of result to send back to the caller of the Web Service. As you can see, it is the same as the hello.

Message
The **HelloMessage** message is the only type of message that will be used with this Web Service. Its content conforms to type **hello** defined earlier.

Operation
This Web Service only offers one operation called **hello** which takes an input, which message’s type is **HelloMessage** and which returns the same type of response.

Binding
This part will link real protocol and data format with the operation defined above. In this example, it binds it to HTTP as a transport protocol, SOAP 1.1 for representing exchanged messages, and the encoding style is document (explained in the next pages).

Port
You can see the address bound to the portType, in this example it is a local server which host the Web Service.
**UDDI**

Universal Description Discovery and Integration (UDDI) is an OASIS standard, which defines a registry for Web services. In short, it defines mechanisms for:

- Describing services
- Discovering business entities
- Integrating business services using the Internet

It is platform-independent framework acting like a phone directory, but specialized for Web services. Service providers use a UDDI registry for publishing the services they provide, classifying them into a category, and describing them technically using their WSDL interfaces.

**SOAP**

SOAP\(^{21}\) is a messaging protocol used by Web services. It is a lightweight, simple, extensible, XML-based protocol used for exchanging structured message information in a distributed environment. Its main advantage over other protocols such as IIOP or RMI is that it was built to work with HTTP\(^{22}\), the most used transport protocol on the Internet, allowing communication behind proxies and firewalls.

![SOAP message layout](image)

Figure 18: SOAP message layout

SOAP defines a set of elements to represent a message:

- The **envelope** is the root element of every SOAP message
- The **header** is an optional element that can contain additional data, such as security related information, routing information…
- The **body** contains the main information of the message, representing the payload of the message.
- The **fault** which represents a specific block contained in the **body** which defines an error that occurred during the interaction.

In Figure 19 you can see an example of a SOAP request.

\(^{21}\) SOAP 1.2 is a W3C recommendation (see (21)).

\(^{22}\) SOAP can be used with other transport protocols such as SMTP, HTTPS, JMS and so on.
Figure 19: SOAP request example

The biggest difference lies in the way you encode your message. There are two ways to encode your data: SOAP encoding with RPC style and literal encoding with Document style. Two ways to encode data to answer two specific needs:

- The need for an application to uses another application’s methods, which is the foundation of RPC itself.
- The need to exchange documents between partners using a standardized format and interpretation of documents and messages.

When using literal encoding, the content of the body conforms to an XML schema, whereas when using SOAP encoded with RPC, the data representation rules uses a subset of XML schema but does not conform to an XML schema. Therefore, the encoding style can be one of the following:

- RPC - encoded
- RPC- literal
- Document- encoded
- Document- literal
- Document-literal wrapped

A detailed explanation on the various encoding styles can be found in Appendix B.
Web Services Stack

To quote one of my lecturers, Mikhail Matskin, ‘Web Services is a soup’. A soup comprised of more than 70 Web Services standards specifications. Web Services specifications referred collectively as to ‘WS-*’ or WS-Constellation to underline the fact that they are a lot of them.

All these specifications are made to support services interoperability. They are concerned with transactions (WS-Transaction), reliable messaging (WS-ReliableMessaging), security (WS-Security) and so on. These are just specifications; it does not mean that they have all been implemented or that you have to use them all when developing or using Web Services. You just use what you need.

<table>
<thead>
<tr>
<th>Business Domain Specific extensions</th>
<th>Various</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Management</td>
<td>WSDM, WS-Manageability</td>
</tr>
<tr>
<td>Provisioning</td>
<td>WS- Provisioning</td>
</tr>
<tr>
<td>Security</td>
<td>WS-Security</td>
</tr>
<tr>
<td>Security Policy</td>
<td>WS-SecurityPolicy</td>
</tr>
<tr>
<td>Secure Conversation</td>
<td>WS-SecureConversation</td>
</tr>
<tr>
<td>Trusted Message</td>
<td>WS-Trust</td>
</tr>
<tr>
<td>Federated Identity</td>
<td>WS-Federation</td>
</tr>
<tr>
<td>Portal and Presentation</td>
<td>WSRP</td>
</tr>
<tr>
<td>Asynchronous Services</td>
<td>ASAP</td>
</tr>
<tr>
<td>Transaction</td>
<td>WS-Transactions, WS-Coordination, WS-CAF</td>
</tr>
<tr>
<td>Orchestration</td>
<td>BPEL4WS, WS-CDL</td>
</tr>
<tr>
<td>Events and Notification</td>
<td>WS-Eventing, WS-Notification</td>
</tr>
<tr>
<td>Multiple message Sessions</td>
<td>WS Enumeration, WS-Transfer</td>
</tr>
<tr>
<td>Routing/Addressing</td>
<td>WS-Addressing, WS-MessageDelivery</td>
</tr>
<tr>
<td>Reliable Messaging</td>
<td>WS-ReliableMessaging, WS-Reliability</td>
</tr>
<tr>
<td>Message Packaging</td>
<td>SOAP, MTOM</td>
</tr>
<tr>
<td>Publication and Discovery</td>
<td>UDDI, WSIL</td>
</tr>
<tr>
<td>Policy</td>
<td>WS-Policy, WS-PolicyAssertions</td>
</tr>
<tr>
<td>Base Service and Message Description</td>
<td>WSDL</td>
</tr>
<tr>
<td>Metadata Retrieval</td>
<td>WS-MetadataExchange</td>
</tr>
</tbody>
</table>

Figure 20: Web Services stack\(^{23}\)

\(^{23}\) A newer version containing other specifications related to Web Services can be downloaded at (84).
2.2.5. Differences between components and Web services

The first three technologies are based on remote methods invocations, whereas the following one is based on a message exchange mechanism.

**Loose coupling**

All of the above paradigms are based to a certain degree on components. Web services are more decoupled than pure components-based systems, that is, they are less dependent on other system; they are self-contained pieces of software.

In component-based systems, the connections between different components are static whereas in Web Services, the different parties are bound together at runtime.

**Standardization**

Web services also use widely accepted open protocol standards such as HTTP, and XML-based data description standard such as SOAP for exchanging messages over HTTP, therefore aiming at a maximum interoperability.

On the other hand, OMG’s CORBA uses IIOP, Java’s Sun JRMP and Microsoft’s ORPC are generally faster than their XML homologue because less they are less verbose, so less processing is required resulting in quicker communication but all these technologies are vendor-dependent.

**Modularity**

Web Services rely on open standards, using a uniform interface, theoretically allowing services to be linked together more easily, ignoring the underlying implementation languages, operating systems,…

**Cross-domains**

One of the biggest advantages is the fact that since SOAP can use HTTP as a transport protocol, it works well with networks protected by firewalls, whereas IIOP or RMI face some issues because they are usually filtered and most of the time blocked by firewalls.

**Ownership**

Services are consumed, they are not owned by the end consumer. They use services on a pay per use basis whereas for components, you can easily buy and install them without any problem therefore owning them.

**Granularity**

Objects and components built from objects tend to be too fine-grained, too small and granular in comparison with Web Services which should be coarse-grained.

---

24 *Loose coupling* is the concept of reducing systems dependencies. This concept aims at avoiding collateral and potentially harmful effects when a part of the system is modified, otherwise modifications can be too risky and the system is vulnerable to failures.
You might wonder “what is the real difference between CORBA and Web Services?” Both are fully capable of linking heterogeneous systems, but CORBA requires your underlying application to be object oriented whereas Web Services don’t really rely on a specific programming paradigm. You can use simple scripting, procedural or functional languages to create Web Services.

The OO paradigm plays an important role in developing Web Services, and software vendors such as Sun or Microsoft perfectly understood that by modifying their server-side platforms to better support Web Services development. Sun did it by simplifying their EJB programming model, which can be used in conjunction with specific annotations to create Web Services with Sun’s Web Services APIs JAX-WS and their Web Services Interoperability Technology (WSIT) project. OO as a basis for creating Web Services is not more important than other paradigm, and when you decide to use OO, you should be careful that you will not end up using SOAP wrappers around existing components and call them Web Services.

Web Services share also other similarities with components and objects which are widely accepted, but sometimes they are perceived differently:

- A service is a unit consisting of data and operations over the data. This can be true for components or objects.

- A service has its own process. This on the other hand may not be true for components nor for objects. For instance, a component can be controlled by a process which handles other components and/or objects.

- A service can interact with other services. In a component-based system and in objects, interactions may have other meanings because they may not have their own processes.

- The execution of a service is seen as being an atomic execution, that is, either it is a complete success either nothing happens.

- There is no real-time requirement over the execution of services.

---

25 This point is arguable. A Service Level Agreement (SLA) can bind the execution of a service and the response delivery to a certain time frame. But for real-time messaging, it truly depends on the transport protocol. For instance the Real Time Messaging Protocol (RTMP) is a proprietary protocol which allows real-time communications over the internet. Some vendors can endorse some real time guarantees, some don’t. An SLA may also specify levels of availability, performance, serviceability…
Chapter 3

“And, all these things need to be coordinated; so we all need to work together, have timers going and everything so we're all coordinated and get this piece of orchestration done.”
- Duane G. Carrey

3. Web Service Compositions

3.1. WS-BPEL

Web Service Business Process Execution Language (henceforth ‘WS-BPEL\(^{26}\)) is an XML-based, standard orchestration\(^{27}\) language for Web Services from the OASIS consortium. It defines a behavioral model for creating flow of actions based on Web Services. It is a business process modeling language aiming at *programming in the large*\(^{28}\), which refers to code which deals with high-level state transitions logic of a system. WS-BPEL comes from two technologies; one from IBM, Web Services Flow Language (WSFL) which is a graph-based language based on XML, and the other is XLANG from Microsoft, which is a block-structured language.

For instance, let’s suppose you are planning to create a Web application using services in Figure 21. A composition might involve the Availability service that will check for available flight and then the Fees Logic to get the price of the tickets...

All these self-contained autonomous Web Services can be coordinated in order to form a new composite Web Service using WS-BPEL playing the role of a central coordinator\(^{29}\).

![Figure 21: The need for Web service composition](image)

\(^{26}\) Latest specification at the time of writing: WS-BPEL 2.0, see (24).

\(^{27}\) Also known as composition

\(^{28}\) Opposed to *programming in the small*. This kind of programming language generally deals with lower-level short-lived behavior such as database querying, files access…

\(^{29}\) The other approach focuses on a decentralized behavioral model where services act as interacting peers without any centralized control. This approach referred to as Web Service Choreography.
WS-BPEL allows you to do create a process flow as illustrated in Figure 22, which can be available to remote clients as a Web Service.

WS-BPEL links to other core specifications like shown in Figure 23.

- **WSDL 1.1**
  
  - Since WS-BPEL defines a Web Service, its interface(s), operation(s)...are defined using a WSDL interface. Also, when a process interacts with other external services, the interactions are made possible through their relative WSDL interfaces that are imported by the process.

- **XML Schema 1.0**
  
  - The variables that are used within a WS-BPEL process are XML-based, and their type(s) is/are defined in an XML schema.

- **XPath 1.0**
  
  - XPath is an expression language used to query XML documents. It is used in a WS-BPEL to access and modify variables, evaluate conditions…

- **XSLT 1.0**
  
  - The XSL Transformation language (XSLT) is used to transform an XML document into another one so to conform a XML schema. For instance, if the WS-BPEL process receives a XML document from one service and has to send it to another service with a different XML schema, it has to be converted. Such conversion can be accomplished with XSLT.
WS-BPEL concepts can be applied in one of the two different ways: Abstract or Executable.

An Abstract process defines a business protocol between parties, revealing only the message exchange behavior. They do not cover all details of the process execution.

On the other hand, an Executable process does provide the internal behavior structure details. Every time I talk about a BPEL process in this thesis, you can assume it is an executable one unless stated otherwise.
3.1.1. Initial example: the *purchase order* process

In order to introduce the concepts used within a BPEL, I will start by presenting an example of a business process. This example is the one used in (17).

Figure 24 represents a UML activity diagram representation of a *purchaseOrder* BPEL process. Let’s assume the following scenario:

When receiving a purchase order from a customer, the process will trigger three concurrent activities; *Initiate Price Calculation*, *Decide on Shipper* and *Initiate Production Scheduling*. Even though the activities are running concurrently, some synchronization is needed. In the above example, in order to complete the price calculation, the shipper price is required, and in order to finish the production schedule, the shipping date is required. Once all the activities have been terminated, the invoice is sent to the customer.
This *purchaseOrder* process interacts with three external Web Services: *shipping*, *invoicing* and *scheduling*.

Since a BPEL is itself a Web Service, the interfaces are defined using WSDL. In Figure 26 you can see the WSDL interface for the process. On the left, a graphical representation of the different operations the process provides and on the right its XML representation.
Figure 26: WSDL interface of the purchaseOrder BPEL process and its XML representation
In the above picture, you can see a graphical representation of the process. It starts with a receive\textsuperscript{30} activity that, as stated earlier, waits for an incoming purchase order. You can also see the three distinct concurrent sequences using the flow\textsuperscript{30} activity. The purple arrows depict the links\textsuperscript{30} activities used for synchronizing activities. The invoke\textsuperscript{30} activities are used for interacting with external Web Services in the form of partnerLinks and finally the reply\textsuperscript{30} activity used to send a reply to the customer.

A fault handler\textsuperscript{30} is also present at the top of the process. It triggers a reply activity with a special message in case an error is caught during the process execution.

\textsuperscript{30} Will be discussed in the next pages
A simple example of a WS-BPEL process for handling a purchase order.

```xml
<bpws:partnerLinks>
  <bpws:partnerLink myRole="purchaseService" name="purchasing" partnerLinkType="lns:purchasingLT" />
  <bpws:partnerLink myRole="invoiceRequester" name="invoicing" partnerLinkType="lns:invoicingLT" partnerRole="invoiceService" />
  <bpws:partnerLink myRole="shippingRequester" name="shipping" partnerLinkType="lns:shippingLT" partnerRole="shippingService" />
  <bpws:partnerLink name="scheduling" partnerLinkType="lns:schedulingLT" partnerRole="schedulingService" />
</bpws:partnerLinks>

<bpws:variables>
  <bpws:variable messageType="lns:POMessage" name="PO" />
  <bpws:variable messageType="lns:InvMessage" name="Invoice" />
  <bpws:variable messageType="lns:shippingRequestMessage" name="shippingRequest" />
  <bpws:variable messageType="lns:shippingInfoMessage" name="shippingInfo" />
  <bpws:variable messageType="lns:scheduleMessage" name="shippingSchedule" />
</bpws:variables>

<bpws:faultHandlers>
  <bpws:catch faultMessageType="lns:orderFaultType" faultName="lns:cannotCompleteOrder" faultVariable="POFault">
    <bpws:reply faultName="bpws:cannotCompleteOrder" operation="sendPurchaseOrder" partnerLink="purchasing" portType="lns:purchaseOrderPT" variable="POFault" />
  </bpws:catch>
</bpws:faultHandlers>

<bpws:sequence>
  <bpws:receive createInstance="yes" operation="sendPurchaseOrder" partnerLink="purchasing" portType="lns:purchaseOrderPT" variable="PO">
    <bpws:documentation>
      Receive Purchase Order
    </bpws:documentation>
  </bpws:receive>
  <bpws:flow>
    A parallel flow to handle shipping, invoicing and scheduling
    <bpws:documentation>
    </bpws:documentation>
    <bpws:link name="ship-to-invoice" />
    <bpws:link name="ship-to-scheduling" />
    <bpws:sequence name="Sequence_shipping">
      <bpws:assign validate="no">
        <bpws:copy>
          <bpws:from><![CDATA[$PO.customerInfo]]></bpws:from>
        </bpws:copy>
      </bpws:assign>
      <bpws:invoke inputVariable="shippingRequest" name="Invoke-request_shipping" operation="requestShipping" outputVariable="shippingInfo" partnerLink="shipping" portType="lns:shippingPT">
        <bpws:documentation>
          Decide On Shipper
        </bpws:documentation>
        <bpws:sources>
          <bpws:source linkName="ship-to-invoice" />
        </bpws:sources>
      </bpws:invoke>
    </bpws:sequence>
  </bpws:flow>
</bpws:sequence>
```
<bpws:invoke operation="sendSchedule"
    partnerLink="shipping" portType="lns:shippingCallbackPT"
    variable="shippingSchedule">
    <bpws:documentation>Aranz Logistics</bpws:documentation>
</bpws:invoke>

<bpws:receive operation="sendSchedule"
    partnerLink="shipping" portType="lns:shippingCallbackPT"
    variable="shippingSchedule">
    <bpws:documentation>Aranz Logistics</bpws:documentation>
</bpws:receive>

<bpws:sequence name="Sequence_invoicing_service">
    <bpws:invoke inputVariable="PO"
        name="Invoke-init_price_calculation"
        operation="initiatePriceCalculation" partnerLink="invoicing"
        portType="lns:computePricePT">
        <bpws:documentation>Initial Price Calculation</bpws:documentation>
    </bpws:invoke>

    <bpws:invoke inputVariable="shippingInfo"
        name="Invoke-send_shipping_info"
        operation="sendShippingPrice" partnerLink="invoicing"
        portType="lns:computePricePT">
        <bpws:documentation>Complete Price Calculation</bpws:documentation>
    </bpws:invoke>

    <bpws:receive operation="sendInvoice"
        partnerLink="invoicing" portType="lns:invoiceCallbackPT"
        variable="Invoice" />
</bpws:sequence>

<bpws:sequence name="Sequence_scheduling">
    <bpws:invoke inputVariable="PO"
        name="Invoke-production_scheduling"
        operation="requestProductionScheduling" partnerLink="scheduling"
        portType="lns:schedulingPT">
        <bpws:documentation>Initiate Production Scheduling</bpws:documentation>
    </bpws:invoke>

    <bpws:invoke inputVariable="shippingSchedule"
        name="Invoke-send_schpping_schedule"
        operation="sendShippingSchedule" partnerLink="scheduling"
        portType="lns:schedulingPT">
        <bpws:documentation>Complete Production Scheduling</bpws:documentation>
    </bpws:invoke>

    <bpws:reply operation="sendPurchaseOrder"
        partnerLink="purchasing" portType="lns:purchaseOrderPT"
        variable="Invoice">
        <bpws:documentation>Invoice Processing</bpws:documentation>
    </bpws:reply>
</bpws:sequence>

<bpws:reply operation="sendPurchaseOrder"
    partnerLink="purchasing" portType="lns:purchaseOrderPT"
    variable="Invoice">
    <bpws:documentation>Invoice Processing</bpws:documentation>
</bpws:reply>

<bpws:sequence>
    <bpws:invoke operation="sendInvoice"
        partnerLink="invoicing" portType="lns:invoiceCallbackPT"
        variable="Invoice" />
</bpws:sequence>

<bpws:reply operation="sendPurchaseOrder"
    partnerLink="purchasing" portType="lns:purchaseOrderPT"
    variable="Invoice">
    <bpws:documentation>Invoice Processing</bpws:documentation>
</bpws:reply>
3.1.3. WS-BPEL 2.0 language constructs

<table>
<thead>
<tr>
<th>Some WS-BPEL 2.0 activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;receive&gt;</td>
</tr>
<tr>
<td>&lt;throw&gt;</td>
</tr>
<tr>
<td>&lt;sequence&gt;</td>
</tr>
<tr>
<td>&lt;forEach&gt;</td>
</tr>
<tr>
<td>&lt;compensate&gt;</td>
</tr>
<tr>
<td>&lt;extensionActivity&gt;</td>
</tr>
</tbody>
</table>

Table 1: WS-BPEL 2.0 activities

BPEL concepts include different building blocks in order to define a behavior. In Table 1, you can see some of the activities the language offers. While some may sound trivial, others have to be slightly explained.

Detailed explanations about the various constructs can be found in Appendix B.
Chapter 4

4. Test automation

“The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.”

- Bill Gates

Testing is an activity that requires huge amounts of efforts and time that the development teams do not have due to the budget and timing constraints. Therefore, reducing costs without reducing the quality of the software is critical.

4.1. Goals of test automation

Test automation generally refers to the use of software for:

- designing tests
- automating and controlling the test execution
- comparing actual results with predicted ones
- reporting test results

Figure 29: Steps involved in a test automation infrastructure
The goals of test automation are to alleviate developer’s burden of tedious, manual testing and improve the overall quality of the software by preventing defects prior the delivery of the software.

As pointed out in (18), building and automating test suites costs money. Sometimes it is worth the money and helps improving the quality; sometimes it worsens everything, as described by James Bach in (19).

In Figure 30, you can see the efforts spent on automating tests are not negligible, and passed a certain hump you will be rewarded by saving some efforts later on. The initial increasing curve until the hump represents the cost of learning new tools, new technologies and new practices. This underlines that nothing comes for free in this world, but the efforts are compensated by reduced costs.

On the other hand, in Figure 31, if you see that the efforts you have put into automating tests are still increasing over time, it generally means that you should reconsider your approach to test automation.

Figure 30: Results of a GOOD test automation strategy. (Courtesy of Gerard Meszaros, used with permission)

Figure 31: Results of a BAD test automation strategy. (Courtesy of Gerard Meszaros, used with permission)
Some high-level objectives of the goals of having test automation are pointed out by the author in (18):

- tests should help us improve the quality
  o We can use test to ensure that we are building the right software, thus reflecting the SUT’s (correct) usage. In a Test first approach such as TDD\(^{31}\), the test actually drives the way to develop the software just by making sure that what we are actually coding make the test passes. Tests are used to avoid the introduction of new defects (used in a regression testing campaign) and to locate them in case errors occur in the tests.

- tests should help us understand the SUT
  o Test show us how the system is supposed to react. They act as documentation, telling us what the result should be. They tell the tester how the system is expected to behave given some input stimuli and what the output events of the system are.

- tests should reduce (and not introduce) risk
  o According to Michael Feathers in (20), code without tests is *legacy code*. Tests play a *safety net* that increases the level of confidence we have in the code. When we inherit someone else’s code, we tend to be reluctant to change his/her code because we are afraid of the (potentially disastrous) effect it may have. When the same code is bundled with tests, we can start to change pieces of code and see which tests fail and more importantly *why* they failed. We should also avoid to introduce specific tests code and leave any test logic outside the system we are testing.

- tests should be easy to run
  o Test execution should be straightforward and easy to do. They should run without any manual intervention. Tests should also report that its expected results are correct (using *Assertions* usually present in xUnit libraries). They should also output the same results now and forever once they are run multiple times in a row. If the tests we write used some shared variables or external data, we must ensure to clean up the side effects after their execution.

- tests should be easy to write and maintain
  o Test should not convey complexity, otherwise they will be too hard to maintain in the long run. We do not want to end up testing our tests or worse debugging them. Simplicity is the key word when creating a test. It should be easy to read and easy to write. They should test one functionality of the SUT at a time, and they should not overlap each other, that is, we should avoid duplication among tests. We should also apply the separation of concerns: testing the business logic and testing the user interface are two separate concerns and their relative tests should be separated as well.

- tests should require minimal maintenance as the system evolves around them
  o *Change* is the only constant in software engineering. We should ensure that the automated tests do not make change in the SUT more difficult. Trying to isolate the SUT from its environment, ensuring the verification of one condition per test will help us minimizing the impacts of changes, and will also reduce the maintenance efforts in the long term.

\(^{31}\) *Test Driven Development (TDD)* is a software technique concept of Extreme Programming, where the test of a functionality is written (possibly) before its implementation. More details can be found in (86).
4.2. Manual testing and automated testing

Manual testing is seen as bad or a waste of time by lots of people compared to automated testing, weaving the fact that computer are faster, cheaper and more reliable for testing than humans are. Manual testing is a process that can cope with change in software more rapidly than automated tests, and thanks to our human mind, we are able to detect several problem patterns just peeking at the code. For small programs that are not meant to evolve over time, that is, which are not going to have several releases in their life, manual tests can be less costly than the time and the money required automating them.

On the other hand, automated tests can provide a guarantee that our tests are made to be repeatable, which can be substantial when it comes to regression testing. They are also very useful when it comes to automatically populate a database for instance. Some specific kind of testing such as performance and load testing cannot run without some automation. Only test automation can help testers who want to cover all possible combinations in a timely manner (even though it’s impossible in certain cases) when facing large systems comparisons of hundred of thousand LOC. Usually, small manual tests involve tedious repetitive tasks that can be automated and speed them up greatly. If maintaining tests is expensive, test automation is a way of ensuring that the costs are kept as minimal as possible.

4.3. When should we automate tests?

There are some cases where automating tests do not give much rewards, and, actually, it’s the total opposite. Based on the following criteria, you can decide to avoid automating your tests if:

- Your GUI is likely to change very often
- Tests are not likely to be repeated
- Your tests cannot see what you (human) can see
- There will be only few releases of the software
- The tool cannot see parts of the SUT or those parts need special programming
- Testers have poor expertise
- A completed end-to-end use case cannot (yet) be implemented by a test
- …

This list is non-exhaustive of course, but it may help you thinking about the usefulness of having automated test instead of manual ones.

On the other hand, here are some reasons why and when you should consider automated tests:

- Test data is needed by tests, the initialization (called ‘setup’) of the fixture 32 and the cleaning of the fixture (called ‘teardown’), for instance, closing DB connection, deleting test data from the DB…
- Tests should be run quickly between releases or they have to run overnight
- End-to-end load tests is needed
- Parts of a manual test is tediously repetitive
- The software has more than 3 releases
- End-to-end use cases can be implemented and need some repeating.
- When the number of possible inputs is large
- Some features need meticulous testing
- …

32 The fixture is defined by the author of (25) as “all the things we need to have in place to run a test and expect a particular outcome”. In few words, it is the data, the server connection, the database connection…that is needed by a test to execute.
An automated test should have the following desired features listed in (21):

<table>
<thead>
<tr>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concise</td>
<td>as simple as possible and no simpler</td>
</tr>
<tr>
<td>self-checking</td>
<td>test should report its own result without human intervention</td>
</tr>
<tr>
<td>repeatable</td>
<td>test can be run many times in a row without human intervention</td>
</tr>
<tr>
<td>robust</td>
<td>test produces same result now and forever despite changes of the external environment</td>
</tr>
<tr>
<td>sufficient</td>
<td>tests verify all the requirements of the SUT</td>
</tr>
<tr>
<td>necessary</td>
<td>everything in each test contributes to the specification of desired behavior</td>
</tr>
<tr>
<td>clear</td>
<td>every statement is easy to understand</td>
</tr>
<tr>
<td>efficient</td>
<td>tests run in a reasonable amount of time</td>
</tr>
<tr>
<td>specific</td>
<td>each test failure points to a specific piece of broken functionality</td>
</tr>
<tr>
<td>independent</td>
<td>each test can be run by itself or in a suite of an arbitrary set of tests in any order</td>
</tr>
<tr>
<td>maintainable</td>
<td>tests should be easy to understand, modify and extend</td>
</tr>
<tr>
<td>traceable</td>
<td>to and from the code it tests, and to and from the requirements</td>
</tr>
</tbody>
</table>
Chapter 5

“Once the races begin it’s more difficult and there is never that much time for testing.”

- Valentino Rossi

5. Testing Web Services and composite Web Services

The dynamic and distributed nature of Web Services yields a unique challenge in testing that is not present in other computing architecture. In component-based system, the test team knows all the components and their dependencies therefore they can methodically plan tests for the system and check its behavior. In an SOA infrastructure, a service can dynamically and non-deterministically be invoked after a registry lookup and be integrated in a service-based application, thus it is be impossible to know in advance which services will form the final application.

Another issue concerns the lifecycle of a service. A service is maintained and changed in an undeterministic and independent fashion; the vendors in charge of the service they provide should be ready to validate it every time a change is made. Sometimes it becomes tricky to test services because service providers charge on the usage, like Amazon S3, therefore it may be too expensive to test those services.

A Web Service composition relies on external Web Services, so each service involved in the business process has to be methodically tested in isolation before it should take part in a composition. After each service has been tested, the composition itself should be tested too in order to guarantee that the composite application is reliable.

Let’s just take a small example. Imagine an application, made out of several Web Services just as illustrated in Figure 32. Each service has a pretty good reliability, but the overall reliability of this application being the product of the individual service is:

$$R_{\text{application}} = \prod_{i=1}^{N} (S_r_i) = 0.90 * 0.94 * 0.96 * 0.98 * 0.89 * 0.97 * 0.90 * 0.95 * 0.93 \approx 60.70\%$$

This can potentially lead to a substantial collapse of the application even though each service has a pretty good reliability. This little over simplified calculus does not even take into account other important variables such as:

- service maintenance downtime
- network partitions

Figure 32: Application based on 'reliable' services
As mentioned in (22) and in many other journals and papers, what makes Web Services testing so special can be narrowed down to four general key issues:

- Lack of controllability

  Components and libraries are physically integrated in a software system, but for services it is not the same, they run on independent infrastructure, and evolve on the sole control of the service provider. So when using the underlying out-of-control services, you cannot really adopt a strategy to evolve your own application or to launch a regression testing campaign.

- Lack of observability of the service’s code and its internal structure

  Obviously, you don’t know anything about the service but its interface. This prevents the use of white box testing techniques because they require knowledge of the structure of the code and the data flow.

- Adaptiveness and dynamicity

  For traditional software, the tester can know which component is invoked in a given call-site, or at least the potential targets. In an SOA infrastructure, a system can be described by a workflow branching together abstract services which are automatically bound to concrete services retrieved by a registry during the execution of the workflow for instance.

- Cost of testing

  As I pointed out earlier, some service providers charge you on a per-use basis, so, knowing that each request is going to cost you some money, you will think twice before executing your tests. Moreover service providers protect their services against massive incoming requests so to avoid a denial of service phenomenon, repeated service invocation may not be applicable, not to mention that a service can produce side effects you don’t want, such as booking a flight ticket in the case of a flight reservation service.
At this point, you may wonder which approach is best suited for testing Web Services, black box or white box.

- **Black box testing**

  Apparently a black box approach for testing Web Services fits the picture perfectly since we have no knowledge of the internals of the systems. Typically, a tester interacts with a system through its user interface by providing inputs and examining outputs without knowing where the results come from and how they were processed. The SUT is exercised over a range of inputs, and the outputs are observed for correctness. Web Services do not have user interface per se, but they do have an interface which you can use to communicate with the system.

  According to (23), black box testing has the following advantages and disadvantages:

  **Advantages**

  - *Efficient* - Black box testing is well suited for large piece of code or for vast amount of units.
  
  - *Unbiased* - We have a clear separation between the user’s perspective and the developer’s perspective. It focuses on the quality assurance part and not on the development responsibilities.
  
  - *Non intrusive* - This technique does not need to access the code.
  
  - *Easy to execute* - This technique can be used by moderately skilled testers who do not have any knowledge in programming languages, networks...

  **Disadvantages**

  - *Blind coverage* - Since we don’t know what’s going on behind the scene, we cannot target parts of the code which may contain more defects than others.
  
  - *Localized* - We only target a limited portion of the code because only a fraction of possible inputs are tested.
  
  - *Inefficient test authoring* - Without structural information, exhaustively testing the SUT will take a large amount of time and will require huge amount of resources.

  This technique is best suited for quick Web Service prototyping and for rapid test scenario testing. It provides a rapid feedback on the functional behavior of the service’s operations readiness. It is also best suited for operations which have tightly defined input ranges that can be found in WDSL’s inline schema for instance thus narrowing down the input domain.
- White box testing

In this case, testers have access to the source code and are aware of the structure of the Web Service. Clearly, they can design test cases so to target code paths in order to achieve a certain level of coverage. This leads to a more structured approach, and helps tester to craft test cases which are more likely to reveal defects.

Authors in (23) identify the following pros and cons when using a white box approach to test Web Services:

### Advantages

- **Increased effectiveness** - testers can crosscheck design decisions and assumptions against the source code therefore yielding a robust design.

- **Full code pathway capable** - All different paths can be tested thoroughly.

- **Earlier defect identification** - Testers can analyze the source code while it’s being written, so they can design test cases earlier and thus find errors quickly.

- **Hidden code flaws revelations** - access to source code can reveal design issues of a system.

### Disadvantages

- **Scaling issues** - It requires a deep knowledge of the system’s environment, testing tools, coding languages, modeling... It requires also experienced and skilled testers.

- **Hard to maintain** - Because of the underlying complexity of the methods and tools used, such as code analyzers, path traversal algorithms, fault injectors…

- **Cultural stress** - the line between the tester and the developer begins to blur and may become a cultural stress.

- **Highly intrusive** - this method may require modification to the source code, or changing it.

White box testing is best suited for Web Services in their early development stage where the tester and the developer can work together to find defects. However white box testing is problematic for a large SOA because a service can invoke a third-party Web Service outside the tester and developer’s control thus lacking knowledge about the service’s programming language, operating systems...Distributed Web Services add access challenges making white box testing across SOA next to impossible.
- **Grey box testing**

This method refers to the technique of testing a system with a limited knowledge of the internal of the system. Testers have access to information which goes beyond simple requirements documents, and with that information they can generate tests. For instance, in the WSDL file, you have information about the data types, the transport protocol… which you can use to create intelligent, efficient and highly target test cases.

Again, authors in (23) come up with pros and cons:

**Advantages**

- *Combined benefits* - testers can combine black box and white box techniques and leverage relative strengths.

- *Non intrusive* - the source code is not modified since tests aren’t based on it. Instead, tests are based on the interface definition, functional specification and the application’s architecture.

- *Intelligent test authoring* - based on the limited information testers can get, they can author intelligent test scenarios, emphasizing on data type handling, communication protocol and exception handling.

- *Unbiased* - the separation between the tester and the developer is maintained. The handoff is only around the interface definitions, documentation.

**Disadvantages**

- *Partial Coverage* - coverage can only be partial without access to the source code and only limited paths can be covered by tests deduced through available information. The coverage highly depends on the tester’s skills.

- *Defect identification* - knowing where the defects are located is inherently difficult in distributed systems. The grey box approach depends on how well exceptions are caught and propagated through the system.

The inherent distributed nature of Web Services, along with the lack of source code access makes white box testing impossible. But still, with WSDL as the de facto standard representation of a contract between the service consumer and the service provider, testers can make use of the ‘rich’ information to construct and automate such tests in order to improve Web Services deployments.
Although one cannot really use a pure white box approach to test Web Services, one can freely use its concepts to increase the effectiveness of the testing process. Intelligent and efficient tests can be created and executed to determine defects based on the information available in the WSDL interface. One can easily understand the need for further automation in testing, and this applies also for Web Services testing. There exists a strong need to automate the test generation process based on available information to increase the testing efficiency. One can easily think of generating test data based on the XML Schema, in order to create SOAP messages which may be valid or not so to test the Web Service’s robustness. You can apply some concepts from mutation testing\textsuperscript{33}, a method generally applied in white box testing, which, briefly speaking, takes the source code, modifies it so to produce mutants as illustrated in Figure 33. The goal of this technique is to see if a test suite is able to detect the change (i.e. one test fails), and if it does, the mutant is said to be killed. In order to create mutant, you need mutation operators. Here are some examples of mutation operators:

- Replacement of arithmetic operators with another one (/,+,\*,\%,…)
- Replacement of boolean condition (true,false)
- Replacement of Boolean expression with another one (<,>,\geq,\leq)
- Replacement of a variable with another variable declared in the same scope (variable types should be the same)
- …

But in our case, we cannot modify the source code; instead we can use this technique at the data types, data value, message structure or protocol binding level. Obviously other types of mutation operators will have to be defined for data mutation, some of which will be explained in a later section of this chapter.

\textsuperscript{33} Sometimes called mutation analysis
In (24) and further detailed in (25), authors underline the fact that the testing phase of a Web Service should be tackled at different levels: unit, integration, system testing.

BPEL and Web Services are intrinsically linked since a BPEL process is exposed as a Web Service, but even though they share a lot in common, they also have differences when it comes to testing. Unit testing a Web Service and unit testing a BPEL process are two completely different things. A Web Service can be considered as a black box system, its internal logic is not known and the only thing we can rely on is its WSDL interface. Tests for a Web Service will therefore be based on that particular interface. On the other hand, a BPEL process can be considered as a white box system, and tests are designed based on its internal structure.

Testing at unit-level in a Web Services based SOA infrastructure is testing the smallest unit, i.e. a Web Service. Therefore unit testing a Web Service is a bit like testing its WSDL interface. At integration-level, multiple Web Services are glued together to form a larger composite application. As a result, verifying the execution of BPEL processes themselves is considered as integration-testing. Finally testing the WSDL interface of the BPEL processes can be considered as system-level testing, which can be done using the same techniques as for a single Web Service.

5.1. Unit testing

In this part, we are considering functional test not at the source code level but we will rather refer to ‘testing the service as a unit’. Prior to publishing your service-based application, one has to ensure the correct behavior of every individual service. Starting only from the specification, that is the WSDL interface, and using a grey box testing approach, I will expose the steps that are needed for testing a single Web Service. I will also briefly present some advanced techniques which are still active research fields that can help automating the testing process.

**Steps for testing a Web Service**

![Figure 34: WSDL 1.1 Over-simplified UML metamodel](image)

In the above picture, you can see a partial over-simplified meta-model of WSDL, which show the different parts of a WSDL interface.
In (24), authors advocate the following steps:

- Test all the *portTypes*  
- Test every *operations* on each *portType*  
- Identify and test the equivalence *classes* on all of the data elements provided to each *operations*

Remember that we don’t know what is going on behind the WSDL interface; therefore we have to focus our efforts on its interface specification, and try to make the best out of it in order to automate the process as much as we can.

Generally, testing a Web Service includes the following tasks:

1. Generate the client or the skeleton code for the Web Service
2. Define the test inputs
3. Invoke the Web Service using the client or skeleton code
4. Verify that the actual response is similar to the expected one

Web Services related libraries such as Apache Axis 2 help you with the generation of the client or skeleton code. These pieces of code will be useful if you are planning to reuse the service in your application; in that case the parts of the code that will call the remote service, methods skeleton blocks for calling the operations the service offers… will be generated from its WSDL interface in Java for instance. Although tools for generating client and skeleton code have greatly improved, they have a big drawback: each time the WSDL interface changes, you will have to regenerate the code. The catch is that you don’t know when the interface changes, so you don’t know if your generated code will continue to work. One solution, provided in (26) is to eliminate the generation of the client or skeleton code and invoke the Web Service directly through a generic HTTP client, like the one provided by Apache Commons HttpClient API.

Another important thing is that we want to examine the SOAP messages we will receive from the Web Service. A SOAP message is ‘just’ XML right? But comparing an XML document to another is not the same as comparing two strings together to see if they match.

Imagine two XML nodes that I want to compare. What do you think, are they the same…?

![Two equivalent XML documents](image)

**Figure 35: Two equivalent XML documents?**

Sometimes we can say that they are equivalent, because obviously the only difference is that the empty `<street>` node is represented in different ways, but sometimes it’s more subtle than that. In some cases you don’t want those two XML documents to be the same because you want two messages to have the same structure. Comparing two XML documents is not as easy as checking if characters match. Luckily for us, we can make use of XMLUnit (27), available for Java and .NET. A nice feature of this library is that it extends the JUnit framework (NUnit for .NET), so it is possible to use...
the JUnit test runner to automate the execution of the tests. It also allows you to use XPath expression if you want to select specific values in an XML document in order to compare it to an expected value.

In the previous tasks listing, we can eliminate number 1 by using a generic HTTP client library and number 3 because it directly derives from number 1. For the 4th task, we can use libraries such as XMLUnit to compare XML documents with one another for correctness.

All these above tasks can be fully automated, but what about the second task?

Selecting test inputs can be done manually of course, by choosing data within an equivalent input domain the tester came up with, so the test data can be randomly chosen from the equivalent partition.

Selecting test data in an automated fashion, according to (28), can be done either statically or dynamically. Data generation/selection generally includes two steps:

- Deriving the constraints

  As a matter of fact, from a tester’s point of view, an XML Schema represents the basic rules and constraints on the data used by Web Services to exchange information. One can use that information to generate relevant data which will help us to subdivide the input domain into sub-domains according to the partition testing principle.

- Solving the constraints

  Once the constraints have been derived, they must be solved so to have test data which can be selected from a range of values satisfying the constraints.

This particular part is probably one of the most difficult and requires some efforts because of the complexity of the inherent complexity of the XML Schema as underlined in (29).
Some authors specifically focused their attention on deriving test cases exclusively from the WSDL interface. In Figure 36, you can see on the left part the WSDL specification and on the right parts the different elements that authors in (30) came up with for automating the testing process. I will develop in detail the method they have established.

Starting from the lowest component in the stack, authors in (30)

1. Test Data Generator

In order to generate test data, you will have to analyze the data types of the messages that the Web Service uses. The messages, generally defined by an XML Schema, can be either simple or complex (31). The XML Schema Datatypes defines a set of 19 built-in primitive types such as decimal, string, boolean... The specification also defines built-in derived types such as normalizedString, integer, nonNegativeInteger... Primitives and derived datatypes are what it’s called ‘simple types’. Each simple type has some constraints35, which describe the aspect of a data type. For instance, the type ‘double’ has the following constraining facets: pattern, enumeration, whitespace, maxInclusive, maxExclusive, minInclusive and minExclusive.

35 Called ‘constraining facets’
- Pattern

This constraint defines a regular expression, expressed as a string, which the datatype needs to satisfy. In the example below, you can see that the ‘price’ element has a restriction in the form of a pattern constraint. The pattern contains a small regular expression, ‘\d{3}’, which simply means that the price has to be a 3 digits length decimal number.

```
<element name="price">
  <simpleType>
    <restriction base="decimal">
      <pattern value="\d{3}"/>
    </restriction>
  </simpleType>
</element>
```

Figure 37: Example of a double datatype with a pattern constraint

There are a lot of different regular expressions which can be used for defining a pattern, and one has to take them into consideration when generating test data. In the case of fault-based testing, you can use the constraints in order to generate data which doesn’t satisfy the constraints so to have faulty messages which can set off at worst a fault or at best a fault-handling mechanism to be activated in the Web Service’s implementation36.

- Enumeration

An enumeration basically defines a space of predefined values which you can choose from for this particular datatype.

- Whitespace

This constraint defines the processing that has to be done when dealing with a whitespace in the data. The value for this constraint can be: replace37, preserve38 or collapse39. The value of whitespace for a double data type can only be collapse.

The other constraining facets for the double type, as well as all constraining facets for every datatype can be found in (31).

Authors linked every facet to a facet generator which will generate and return test data depending on its type.

---

36 Actually you have to read: at best a fault and at worst a fault mechanism… since the goal of testing is to reveal defects.
37 tab, line feed and carriage return are replaced by a space.
38 The value is not changed.
39 Contiguous sequences of space will be collapsed to one space.
2. Operation Flow Generator

Generally a service will offer multiple operations, and these operations can be combined so to form an operation flow as coined by authors in (30). With the data that was generated earlier, not only they can generate test case for testing individual operation but also for sequence of operations. They do that by analyzing the dependencies of each operation. For instance, suppose you can only retrieve a trade quote (using a getQuote operation) from a service after successful login (using a login operation), then you know that getQuote is dependent on login. They also use another method for reducing the number of test cases: the input-output dependency analysis. They defined three types of dependencies:

- Input dependency
  
  If operation1 and operation2 share a set of input messages, they are said to be input dependent.

- Input-output dependency
  
  If at least one operation1’s input message is the output of operation2, they are said to be input-output dependent.

- Output dependency
  
  If operation1 and operation2 share a set of output messages, they are said to be output dependent.

They developed an algorithm which will perform these analyses on each operation and identify the order in which the operations have to be positioned into the operation flow. Sometimes the number of operations that has to run before a given operation is too large, and that is why they combined certain dependencies in order to categorize operation flows into groups. The operations in these groups are not combined but rather developed into different test cases.

3. Test Specification

Test cases are encored in an XML format called Service Test Specification (STS) that can be easily exchanged within a testing infrastructure, or even bound to SOAP messages in order to execute them. Authors came up with a schema definition for this STS, which take concepts of WSDL by specifying the operation name, part name, input message…

Authors finally discussed the different coverage criteria they introduced, such as:

- Part coverage
- Message coverage
- Operation coverage
- Operation flow coverage

All these criteria make sure that the generated test cases and test data cover all the parts, message, operation and possibly every operation flows which are based on the dependency analysis.
### 5.2. Integration testing

When it comes to testing BPEL processes, things become trickier. Ensuring a correct behavior of the composition requires knowing its internal logic. For that reason, BPEL testing can be considered as a kind of *white box* testing.

In (24) and also in (32), authors advocate for the following steps to take in order to test a process:

- Determine the path(s) of the process which need to be tested
- Determine the data variation which needs to be tested along the path. This can be done by looking at the possible inputs of the process activities and by ensuring that conditions are properly covered along the path.
- Develop a mapping from the test cases to the elements in the process, along with partners involved and the operation targeted, so when a partner changes, we know which test(s) to re-run.

Testing a language such as BPEL is far from being trivial, because of the complexity of the language, concurrency, fault-handling… as well as related specifications make BPEL processes testing very difficult.

Philip Mayer, in his thesis (33), created a framework for unit testing BPEL compositions, by introducing *BPELUnit*, a new member of the xUnit family aimed at testing BPEL processes. He started with the very fact that few research has been done concerning the unit testing of BPEL to design and implement a new testing framework. I believe his work has paved the way to further dig up white-box testing of BPEL processes so in order to improve the service’s quality.

Authors in (34) took a very different approach. They went down the formal verification road by using a famous model checker, *Spin* (35) (36), to generate test cases specifications for BPEL compositions. In order to do it, they translate the BPEL process into Promela (Process Meta Language), a verification modeling language used in Spin. The language allows you to create concurrent processes to model a distributed system. The properties of a model can be verified using assertions in the model or expressed in Linear Temporal Logic (LTL). The tool will explore all the possible states of the model and check if the properties hold. If the properties don’t hold, Spin provides a counterexample along with the path that lead to the error. This is this particular feature that the authors used so in order to generate test cases specifications for BPEL composition. In brief, they transform BPEL specification into Promela directly, without going through an intermediate representation as other researchers usually do, such as in (37).

Authors in (37) proposed a formal semantics for BPEL, Web Service Automaton (WSA), an extension of Mealy machines. They transform BPEL into WSA and then WSA into Promela. They also translate WSA into NuSMV(38), another model checker. Their WSA model supports the analysis of both BPEL control and data flow and provide semantics for both BPEL internal and external interactions. The generated test cases are then executed on the JUnit test execution engine.

Both authors in (34) and (37) share a common idea: to use model checkers as test generation engines. These two model checkers, Spin and NuSMV, will generate valuable counter examples which violate certain properties in a given model. Ensuring that the tests that are generated will go throughout each possible path, as illustrated in Figure 38, is only one type of coverage criteria that authors in (34) and

---

40 Spin is a widely used tool in formal verification, and probably one of the best. His creator, Gerard J. Holtzmann currently leads the NASA JPL Laboratory for Reliable Software. The tool was awarded the prestigious System Software Award for 2001 by the ACM.

41 In LTL, you can encode formula about the future of paths such as that a condition will eventually be true, that a condition will be true until another fact becomes true…
(37) target. BPEL-based testing, i.e. generating test cases from BPEL compositions is an active area of research.
5.3. Related work

- Commercial Web Services testing tools

Web Services testing is an active research area, but it caught also the attention of several of companies which now propose commercial tools for testing Web Services. Leading companies such as IBM with their Rational Tester for SOA quality (39), HP (after acquiring Mercury) strengthen their strategy on testing tools with their HP Service Test software (40). SOA testing is a core business for other companies such as Parasoft with SOAtest (41), iTKO with their LISA Enterprise SOA Testing (42) and Mindreef with SOAPScope (43).

- Open source Web Services testing tools

Commercial tools are not the only available tools – there are a lot of free and open source tools available for functional and performance testing like SoapUI (44), which adoption is skyrocketing due to its flexibility and its user friendliness, TestMaker from PushToTest (45), WebInject (46) so on and so forth.

Lots of these tools share common libraries under the hood such as Apache Commons HttpClient (47) which provides a generic HTTP client and XMLUnit (27) a JUnit extension which provides means to test XML documents. Using the former two libraries in conjunction with JUnit, authors in (26) presented a simple way to automate Web Service testing using Java.

- TTCN-3 based approach

Other approaches use the Testing and Test Control Notation Version 3 (TTCN-3), a strongly typed scripting language developed by the European Telecom Standards Institute (ETSI) used for conformance testing of communicating systems. The author in (48) presents a framework for testing Web Services using TTCN-3. He made the following contributions:

- A detail specification and an implementation of a mapping between WSDL and TTCN-3
- Some extensions to TTworkbench Basic (49), a commercial full-featured integrated test development and execution environment tool from Testing Technologies.

Authors in (50) also use TTCN-3 to test Web Services. They present a mapping from XML to TTCN-3 as a basis for automated Web Services. They also make use of a tool from Testing Technologies, TTthree.

- Web Services/SOAP monitoring

Some authors highlight the importance of self-checking of a service-oriented system by monitoring. Authors in (22) argue that runtime monitors can help in checking for both functional correctness and the satisfaction of QoS expectations. Monitoring can help testers verifying that correct SOAP messages are exchanged between parties. But real-time monitoring is not sufficient because problems may be discovered too late. Several open source tools such as tcpmon (51), wsmonitor(52) or Apache Axis SOAP Monitor (53) can help monitoring data flowing on a TCP connection. One can also use Apache Synapse, a mediation server, which can be used as a monitor, but with the difference that its flexibility and extensibility allows you to modify SOAP messages using XSLT, XPath (54), XQuery (55), Java or even Groovy. You can think of developing Java classes which can insert faults in

---

42 Apache Synapse is a lightweight enterprise service bus (ESB). WSO2 developed a product called Enterprise Service Bus (87) which is powered by Apache Synapse. They added a powerful user interface and other useful features.

43 XQuery is an XML Query Language.

44 Groovy is a dynamic object-oriented language that runs on the JVM.
SOAP messages to test service’s robustness at runtime. Monitoring can be combined with testing to validate service-based systems.

- Model based testing

Some researchers focus on testing Web Services using formal models. Authors in (56) use information present in WSDL to serve it as an input for Model-Based Testing. They use a variant of state machine, namely Symbolic Transition System (STS), as a formal model of the legal ordering of invocations of a Web Service. As a matter of fact, a WSDL interface does not provide the correct and valid ordering of neither the methods calls nor their influence on each other. They use an algorithm which can help them to generate test cases from an STS specification.

When using formal methods for testing, researchers generally encounter lots of challenges, but the two following main problems have been (and still are) critical:

- The state space explosion problem

Complex state machine based models may take hours or even days to explore. The number of states for $P$ processes each having with $K$ individual states is at most the number of $P$-tuples of $K$ values, that is, $K^P$ according to (57). Exploring large possibly infinite models cannot be done without reducing the state. Several techniques can be used in order to reduce the state space such as:

- Partial order reduction (13)
- Static slicing (58)
- Symmetry reduction (59)
- Abstraction (60)
- …

- The model equivalence problem

One may have serious doubts on how well a model can actually represent the possible behaviors of a Java program for instance. The semantic gap between the model description language and the implementation language may be large and can lead a model checker to find errors in the model which may not be present in the real program. These false alarms (or false warnings) are time consuming and lead testers to possible dead ends. That is one of the reasons why researchers implemented automatic model extraction algorithms to reduce the gap between the model and the actual source code as authors have done with Bandera (61), a tool set for model checking concurrent Java software.

-XML focused

Various researchers focus their attention on XML rather than Web Services itself. Since almost every standard used in Web Services is XML-based, their results can be applied to our domain of interest. Authors in (62) concentrate their efforts on applying mutation testing concepts to XML Schema. They present an XSD mutation system along with a set of XSD mutators. Some of their mutation operators are:

- Target and Default Exchange (TDE)

This operator simply exchanges the XML Schema target namespace and the default namespace.
• Element Namespace Replacement (ENR)

This operator replaces the namespace identifier of an element with another one.

• ComplexType Compositors Replacement (CCR)

This mutation operator replaces complex type compositors. XML Schema defines three compositors: `choice`, `all` and `sequence`.

• ComplexType Order Change (COC)

This mutator changes the order of elements of a sequence.
Obviously the authors come up with more operators. These mutators as you may have understood inject errors in SOAP messages in order to reveal defects in the behavior of the Web Service being tested.

SOAP messages as mentioned earlier can be altered to check if the service is robust to these variations. This technique is known as *data perturbation* or *data variation* and derived directly from the mutation testing method introduced earlier. It can be used when you are designing a category partition for the data you intend to use as inputs for the service’s operations. The process works by altering the request message, resending the modified request message and analyzing the response message for correct behavior as mentioned in (63).
Part 2
Chapter 6

6. Case study: Shopping Dossier Automated Tester (SDAT)

6.1. Background

The *Shopping Dossier* is an Amadeus IT Group middleware conceptual framework which unifies the access to independent business components that communicate with back-end systems. The framework is a service offered to travel agencies, which acts like a *super Shopping Basket*, just like the one you can find in E-commerce Web sites, but with more complex features. The *Shopping Dossier* provides different functional and business logic such as “BookingLogic” (responsible for dealing with booking functionalities), “FeesLogic” (responsible for dealing with fees features), and so on…

The main goal of this framework is to provide transparent access to high-level applications, such as the one which are used by travel agencies to book a trip. This middleware also connects to other back-end systems which provide data persistency, search services, so on and so forth.

6.2. Problem definition

Whenever changes are to be made to the *Shopping Dossier*, the development team, prior to handing over the source code to the Quality and Assurance team, has to run some test scenarios which are given in advance by another team in a informal textual format in order to validate the behavioral aspects of the components within the framework. The goal of a test scenario is to ensure the correct functionalities of the *Shopping Dossier* components.

Because of the nature of the Shopping Dossier, the creation of test scenarios is not trivial because of the following reasons:

- Different independent complex component logics can be involved in one test scenario
- Some conditional logic might be required to simulate real use cases
- A scenario may include different complex data representations
- A scenario can rely on another scenario(s) during its execution

The list of test scenarios is an informal document, generally grouping scenario made of several test steps. These test steps describe a specific flow of actions that are supposed to produce some results, which have to be checked against expected result to ensure it is valid and not wrongly processed by the various components.

The team wants to exercise the components directly without involving other layers such as the user interface or the application layer within the server. In other words, they want to solely focus on the *Shopping Dossier* without taking care of higher-level application layers, such as the service layer or the application as you can see in Figure 46.

The *Shopping Dossier* team wants to automate this process as much as possible, getting rid of tedious activities, in order to produce high quality tests that are going to be used to validate the behavior of the *Shopping Dossier*’s underlying elements.
6.3. Objectives and requirements of the tool

The tool should allow developers (and non-developers in the future) to:

- Create and store complex test scenarios without too many efforts

  There is a need to create powerful scenarios that will reproduce steps derived from higher level layers. These ordered steps have to be bundled so to create an executable flow using the Shopping Dossier business components. All the created flows are required to be stored for further reuse (mainly for regression testing).

- Execute the scenarios and store the data involved

  The test scenarios have to be executable without too many efforts, possibly automated, and should store its related data during its execution until it terminates.

- Report and store the tests results

  After building and running the test scenarios, the tool should produce test results, which should be kept for later review and/or possible comparison with previous results.

Below, in Figure 43, you can see a small example of what a test scenario can be. It retrieves a Dossier (which is the French term for folder) which contains information about a traveler (dossierID, the currency code, the total balance of the trip...) and sets it to be active. This test scenario can be exercised with different input data, that is, it can be given a correct dossier number, and therefore a specific dossier is returned by the system, or it can be given an erroneous dossier number and then an error must be returned (if caught).

Figure 43: Test scenario example
A test scenario has or may have the following properties:

- It has to be stateful, keeping the data values during its execution
- It has to be persisted with its test results
- It may include conditional logic
- It can be re-executed during a regression campaign
- It can take part in other test scenarios’ execution, as pictured in Figure 44.
- It may exercise a remote system, i.e. the SUT can be located in another protected network than the one the developers use when developing

![Diagram of scenarios flow](image)

### 6.4. Expectations

- The tool should be integrated within the developer’s IDE: Eclipse and should not modify the current development platform.

  The *Shopping Dossier* team expects a tool that has to be as unobtrusive as possible, that is, it should integrate well with existing integrated development environment (Eclipse is the main Java IDE at Amadeus), and should target the existing standard development platform within Amadeus possibly without modifying it.

- User friendliness

  The tool should provide some level of user-friendliness for creating, executing and storing the test scenario.

- Predicate mechanisms

  The tool should provide mechanisms to check the correctness of complex structured data involved in a scenario.

- Free and open-source technologies
The tools and technologies that are used for the project should be free and open-source.

You can see in Figure 45 the high level overview of the tool, and the different parts it consists of. You can see that the test execution engine should directly target the Shopping Dossier components without going through other higher layers such as the Application layer or the Service layer as you can see in Figure 46. The tool can make use of a reporting engine in order to format the results according to the tester’s preferences. All the results are to be kept somewhere, either in flat files or in a database.
6.5. *Shopping Dossier* general architecture

Most of the architecture is Java Enterprise-based, and the business components such as *Fees Logic* or *Dossier Logic* illustrated in Figure 46 follow the Enterprise JavaBean (EJB) programming model, residing on a Java application server such as BEA WebLogic.

Each business component in the *Shopping Dossier* framework is a self enclosed independent unit. They don’t directly interact with one another; instead they are composed at a higher level so to form a specific application. This clear separation eases the future evolution of the framework’s components, but can make testing a very difficult process.

*Figure 46: Shopping Dossier, high level architecture*
Chapter 7

7. Case study (SDAT): Investigations and solution proposal

Starting with the fact that the Shopping Dossier will be exposed as a Web Services in the future according to its creators, I decided that I will use Web Services related standards that allows the creation of business flows using external Web Services: Web Services Compositions.

7.1. Creating the scenarios

So, my starting point was Web Services orchestration, but because non-developers might be involved in the process of creating specific flows in the future, I considered a pure business modeling language, that is not directly executable, BPMN, and also BPEL which I presented earlier.

Business Process Modeling Notation (henceforth “BPMN”) is a Business Process Management Initiative (BPMI.org) that released the BPMN 1.0 specifications in 2004. Since March 2006, BPMI and the Object Management Group (OMG) announced the merger of their Business Process Management (BPM) activities. Consequently, BPMN 1.0 specifications were adopted in 2006 by the OMG, which is actually working on version 2.0.

Let’s dig briefly into BPMN now and why I am talking about it.

7.1.1. BPMN or BPEL?

BPMN is a graphical notation (graph-oriented), (whereas BPEL is a block-structured one, even if it provides graph-oriented constructs with some restrictions). The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes.

BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A Business Process Model, then, is a network of graphical objects, which are activities (i.e., work) and the flow controls that define their order of performance.

A BPD is made up of a set of graphical elements that ease the development of process modeling through simple diagrams that are familiar to business analysts.

So it was geared for business analysts in the first place. Unlike BPEL, BPMN is not directly executable, but (and that’s where it is interesting) it can be translated into a BPEL. Some vendors, such as Intalio, provide this feature for free (but not Open source) in their IDE based on Eclipse.

Here is a BPMN process diagram and its BPEL equivalent, which is a simple Hello World program that takes as an input a String and will reply “Hello” with the input string concatenated. For instance, if your input is ‘Francesco’; your output will be ‘Hello Francesco’.
You can see that the BPMN notation is rather clear: you have lane, task, messages, pools...

A pool represents a partner of the process, similar to a PartnerLink in BPEL. It also acts as a container of activities, partitioning the whole process.

A Lane is a partition within a Pool to further partition activities within a process.

A message flow is used to picture a message exchange between two participants of the process.

A task is an atomic unit of activity within a process.

These are just tiny pieces of the BPMN’s features, further details about the complete BPMN’s notation can be found in (64).
In Figure 48, you can see the graphical representation of a BPEL process, which is the equivalent of Figure 47.

So the main question was which one to use, BPEL or BPMN?

- BPMN can be used by non-developers to create test scenarios in an easy way, while developers might find it too abstract because it may not reflect the SUT in details as they would like to.
- BPEL can be used by developers and non-developers to a certain extent, because even if BPEL designer tools do exist and provide users a way to create BPEL processes using graphical elements, it remains a complex language to master.

The best of both worlds would be to have BPMN that will generate its equivalent in BPEL. The semi-good news is that it is feasible to some extent; in BPMN’s specifications in (64), authors proposed a mapping to BPEL4WS. In (65) and (66), authors also tried to translate BPMN in BPEL directly. It is a notorious challenge because BPMN is a graph oriented and BPEL is a block-structure, they represent fundamentally different classes of languages.

Due to the fact that BPMN is still a relatively new standard, good tools and documentation are still on their way but, its adoption increases. BPEL on the other hand, thanks to its wide acceptance in the enterprise world remains a better choice at the moment, that is why it will be selected as a our language for creating our test scenarios.

Figure 49: A test scenario is represented as a BPEL process
So a test scenario is a BPEL process, as pictured in Figure 49, which is used with the underlying Shopping Web Services. Of course it will not be enough because BPEL was not geared to be used as a test language, but in our case it will be used as a template of sequences of actions we are interested in testing. It is an intermediary between the SUT, the Shopping Dossier, and a specific Web Service testing tool. But given the fact that BPEL is a recursive technology, that is, it is exposed as a Web Service, we can take advantage of that like illustrated in Figure 50.

![Diagram](image)

Figure 50: BPEL processes as scenarios templates

BPEL process designers are available, free and/or open sources. The one I was interested in is Eclipse BPEL designer, which is an authoring tool for supporting WS-BPEL 2.0 process creation, validation and so on. This tool comes as a free and open source Eclipse plug-in, thus satisfying the constraints I was given.
7.2. Executing the scenarios

Given our executable processes, we need a specific engine to execute them. This role is fulfilled by a BPEL engine. The BPEL process in our case should be synchronous, that is, when the process is initiated, it runs and sends a reply back to the caller in a single Web service call. The caller will block and wait until a response is received from the terminating process.

Most of the BPEL engines are Web applications residing on a server, generally providing the following features:

- Deployed BPEL processes listing
- Persistency mechanism for all the running process instances and their related data handling
- Versioning mechanism of the processes
- *hot-deployment* of the processes, i.e. you do not need to shutdown and restart the engine when you want to deploy a new process
- Management interface for processes, process instances and messages

Another particularity is that the BPEL scenario template will create new instances of the scenario process each time a request is sent, by setting the `createInstance` attribute of the `Receive` activity to ‘true’. All the process instances are persisted while executing by the engine itself generally to a database with all the related data, that is, the input message, the internal variables values and the reply message. This will allow to fully trace back the data during the execution of a process, and to reconstruct it in order to visualize the data flow and value and check where an error was thrown.

![Figure 51: BPEL engine general operating overview](image-url)
Various BPEL engines are available, and they all come in different flavors. All the major companies backing up the BPEL specification propose their specific BPM suites. Oracle provides a complete solution with their Oracle BPEL Process Manager, and they\footnote{Actually Collaxa, which was a BPM vendor company, did the whole work; Oracle ‘just’ bought the whole company in 2004 (89).} come up with a nice Web application which enables users to directly invoke processes, launch previously built tests suites and check out their results.

But because one of my constraints was to choose a free and open source one; my choices were basically reduced to either ActiveBPEL from ActiveEndpoints or Apache ODE. ActiveBPEL proposes a nice stack, with a nice BPEL process designer and simulator along with their specific engine. Thing is that they impose some limitations on their engine which cannot be deployed on any server we want. Apache ODE is, comparing to other engines, a relatively young engine but was proven to be a strong free and open source alternative, very robust with decent performances, that is why I decided to select it.
### 7.3. Web Service Testing Tool

A Web Service test tool should include all the different components which are specific to Web Service and all its related technologies. It should provide means to invoke Web Services, inspect messages which are sent to and received from those Web Services, create assertions… This is far from being trivial, just consider all the related specifications which are involved in Web Services: SOAP, WSDL, XML Schema, XML, HTTP…

For each of these specifications, you need specific libraries such as:

- XMLUnit (27), for creating assertions on XML documents
- XPath libraries for enabling the evaluation of XPath expressions on SOAP messages
- WSDL4J for parsing WDSL files
- Saxon which provides an XQuery and XSLT processor, for creating specific XQuery expressions on SOAP messages
- Apache Commons HttpClient (47) for invoking the Web Service
- A Java/XML binders such as XMLBeans in order to access XML by binding it to Java types, if you develop the tool using Java

The tool should provide means to structure the tests into test suites, test cases…and possibly it should also allow testers some flexibility in the sense that one should be able to add any relevant data, notes or description with a specific test case. A test tool should somehow and to a certain level relate to the UML Testing Profile (11) in terms of test architecture, verdicts, faults…For instance, the UML Testing profile encourages the following minimal test verdicts: **fail, pass, inconclusive and error**.

- **fail**: means that the SUT does not behave according to what it should be expected
- **pass**: means that the SUT behaves according to what it should be expected
- **inconclusive**: means that we cannot know if the SUT performs well or not
- **error**: tells that the test system itself and not the SUT fails

The line between these test verdicts would further blur in the case of distributed components, since we cannot distinguish a SUT crash from a network partition.

Structuring tests is as important as the test itself; a Web Service test tool should provide a neat organization, the addition of meta-data to tests, logging mechanisms…

Luckily for us (mostly for me), Eviware provides a free and open source tools called SoapUI (44) which is, in my humble opinion, one of the best open source Web Services testing tool available. Not only it allows creating powerful tests but also provides the possibilities to create our own extension using SoapUI APIs so to add the functionality we want. It aims at functional testing, load testing and Web Service simulation and mocking.
7.4. Issues ahead

The main issue I faced is that there was no Web Services for the Shopping Dossier. Discussions about its future architecture were only at their beginning when I came up with the proposal. So I had to come up with a temporary solution to expose some functionalities of the Shopping Dossier so to validate a prototype of the proposed solution.

Given the current architecture, two options were possible to expose it:

- Expose the complete Business Façade
- Expose all the underlying logics as single Web Services

Investigations were carried on how to expose the current J2EE based architecture as Web Services without running into many efforts.

Here is a summary of the investigation concerning the exposition of the Shopping Dossier as a Web Service:

What is needed: all the business logics that the Shopping Dossier Façade provides, i.e. all the methods from all of the business components.

If I expose the Business Façade, I will have to be able to access the needed methods of all the involved EJBs.

If I directly expose the EJBs, it would be easier but I will break the fact that the team wants a single entry point to access the Shopping Dossier.

In order to use Java code in a BPEL process, we can either use Web Services Inspection Language (like WSIF) binding or we can wrap the Java code as a SOAP service. The WSIF binding is the common way of using Java code with a BPEL process. But apparently, we have to write the binding manually.

If I choose for instance to expose each EJBs as a service, this mean that every time that one behavioral aspect of the methods within the EJB is modified, the WSDL interface will have to be modified as well.

As an alternative to WSIF binding, you can wrap the Java code as a SOAP service. As with WSIF binding, this method requires that the Java application have a BPEL-compatible interface. A Java application wrapped as a SOAP service appears as any other Web service, which can be used by many different kinds of applications. There are also tools available for writing SOAP wrappers.

However, a Java application wrapped as a SOAP service has the following drawbacks:

- It loses performance, because interactions are constantly being mapped back and forth between the Java code and the SOAP wrapper.
- It loses interoperability, that is, the ability to perform several operations in an all-or-none mode (such as debiting one bank account while crediting another, where either both transactions must be completed, or neither of them).
One other interesting way of creating Web services is by using the features that Apache Axis2 (henceforth “Axis2”) (67) provides. At this moment, I know that the SOAP implementation the team is using is the Apache Axis in its first version, Apache Axis 1. Axis2, as I said is an SOAP implementation (i.e. a complete re-write of the W3C SOAP specifications). Two different implementations on the top the Axis2 architecture were made: Apache Axis2/Java and Apache Axis2/C. In this project, we are more interested in the Axis2/Java.

After this investigation and after some discussions about the freedom to modify the current development platform, it was decided that no extra library was allowed to be added. Therefore, the WSIF framework was directly dismissed, as long as the Apache Axis2 libraries.

The only solution left was to reuse existing libraries that were used to create Web Services within the development platform, that is, Apache Axis 1.
Chapter 8

8. Prototypes architecture and implementation

Figure 52: Original high-level architecture of the testing ecosystem

Figure 52 you can see the initial architecture I came up with while at Amadeus. It involves the following components:

- The Eclipse BPEL Designer, for creating the BPEL 2.0 processes
- Apache ODE, for executing the processes
- SoapUI, for creating test suites, assertions...aiming at exercising the deployed processes
- An embedded Groovy script which was developed as a test step, with the objective of traversing the whole project structure and creating an XML document that exports the results of the tests.

Here are the general steps for using the prototype:

1. The tester creates a BPEL process\textsuperscript{46} which follows its informal representation given to him/her by the other team.

2. When the BPEL process is created, manually checked for errors, it is deployed on the BPEL Engine as a Web Service. (The deployment can be automated using Ant)

3. A new test project is created in SoapUI, pointing to the WSDL endpoint of the deployed process. The tester can also directly points to the SUT WSDL endpoints, but here the goal is to test scenarios made with BPEL.

\textsuperscript{46} The BPEL process should be synchronous.
4. Test suites, test cases, assertions...are created for the current project.

5. Tests are executed and results analyzed.

At this point, the tester can use the Groovy script I made for exporting the result automatically when the tests are being run. The script will traverse all the different test suites, test cases, assertions, using the SoapUI APIs and output an XML file containing the test results. Because this script is itself a test step, it can be enabled and disabled whenever the tester wants it. In the picture below, you can see a partial view of what the script does export.

Figure 53: Partial view of the exported results in XML
In Figure 54, you can see an evolution of the previous architecture. This was thought and made after the internship at Amadeus. This architecture involves new components which offer new functionalities. These functionalities were developed as proof of concepts.

The new component prototypes of this architecture are the following:

1. SoapUI Extension
2. BPEL Monitoring Web UI
3. SoapWUI

Within those three prototypes, there are other sub components which were also created. Let me further detail their purposes, architectures and implementation details.

Figure 54: New architecture with additional components
8.1. SoapUI Extension

8.1.1. Purpose

The goal of this extension is to enable SoapUI testers to start a Web Service which enables third party tools to:

- Retrieve all the information about the projects, test suites, test cases, test status…
- Run selected tests and return their results

This extension makes use of Axis2 Web Service engine, which can be used as a standalone server, that is, it does not need to be deployed on an application server such as Apache Tomcat. The Web Service can be started and stopped upon request of the SoapUI tester by using the simple GUI I have made. Once the Web Service is started on a chosen port, a Web application can make use of the different functionalities the service offers.

8.1.2. Architecture

In Figure 55 you see the architecture of this extension. This extension embeds the Axis2 Web Service engine and all its dependent libraries. Once the service has started, the functionalities are exposed as a Web Service, which will handle all incoming requests, process them and reply accordingly. All communications between the external world and the Web Service is done via SOAP messages. In order to simplify the configuration of the engine, a GUI is provided to the user. The GUI will help set up the port on which the port will be started. The Web Service will communicate internally to SoapUI through its API.
8.1.3. Implementation details

This part of the tool was implemented using Java. As I said before, for creating the Web Service, I
used Apache Axis2, which is the Web Service stack from Apache, and in my opinion one of the finest.
Axis2 comes with a new XML Infoset representation called AXIOM\(^{47}\), which is based on StAX\(^{48}\); a
Java based API for pull-parsing XML. Apache decided to build the SOAP specification on top of this
new model, since a SOAP message is XML, respecting the SOAP message structure.

All the messages exchanged with the external world are handled by special receivers which transform
the incoming XML messages into an AXIOM Object Model. The Web Service offers the following
methods:

<table>
<thead>
<tr>
<th>Return element</th>
<th>Method name</th>
<th>Parameter(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMElement</td>
<td>getAllProjects</td>
<td>nothing</td>
<td>This method will traverse the current SoapUI workspace and get the projects. A message, containing the different project name is constructed and sent back to the caller.</td>
</tr>
<tr>
<td>OMElement</td>
<td>getProjectTestSuites</td>
<td>projectName</td>
<td>This method will traverse all the test suites, its related test cases, test steps and specific assertions...of the selected project. It will construct a response containing all the various information and return it to the caller.</td>
</tr>
<tr>
<td>OMElement</td>
<td>createNewTestRequest</td>
<td>projectName, testSuiteName, testCaseName, testRequestStepName</td>
<td>This method will create a new blank SOAP request for the test case request step. This request conforms to the WSDL interface of the process being tested, in terms of structure, namespaces...This request is then packed and sent back to the caller.</td>
</tr>
<tr>
<td>OMElement</td>
<td>getProjectFile</td>
<td>projectName</td>
<td>This method, although not used at the moment, will take the whole project file and return it the caller. The reason of the existence of this method is that the project file is an XML based file containing all the information about a project. While sending the whole file back to the caller seems a simpler idea, it is not very efficient in terms of bandwidth, and it is not very flexible.</td>
</tr>
<tr>
<td>OMElement</td>
<td>shakeNewRequest</td>
<td>projectName, testSuiteName, testCaseName, testRequestStepName</td>
<td>This method will first create a new request which conforms to the WSDL interface of the process under test, and then it will modify its structure by changing the namespaces, inverting the nodes’ positions... Then the modified request is sent back to the caller. The goal is to create corrupted SOAP messages to test the robustness of the service.</td>
</tr>
</tbody>
</table>

\(^{47}\) AXIOM: AXIs Object Model

\(^{48}\) StAX: Streaming API for XML
All the different response messages are created using AXIOM. In the figure below you can see the SOAP response you get from the `getAllProjects` method.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:tns="http://th.fb">
  <soapenv:Body>
    <tns:getAllProjectsResponse xmlns:tns="http://th.fb">
      <tns:return>
        <tns:project>
          <tns:projectsCount>5</tns:projectsCount>
          <tns:project>0de</tns:project>
          <tns:project>OUR HELLO WORLD</tns:project>
          <tns:project>OUR Instance Management</tns:project>
          <tns:project>test_hello_edw</tns:project>
          <tns:project>SoapUI5</tns:project>
        </tns:project>
      </tns:return>
    </tns:getAllProjectsResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 56: SOAP response from the `getAllProjects` operation

Below, in Figure 57, you can see the simple configuration GUI which is used inside SoapUI for setting the port number of the Axis2 server, as long with other settings such as the team the tester is in, the CR\(^{49}\) or PTR\(^{50}\) the project is targeting…This UI was implemented in Java, and uses the Swing Component libraries.

![Figure 57: Simple GUI for configuring the embedded Web Service Engine](image)

This extension, as well as the required libraries, is dynamically loaded into SoapUI upon startup, and available in the menu to the tester by right-clicking on a project.

\(^{49}\) **CR**: Change Request, is a specific number used within Amadeus that relates to a new functionality which is being implemented by a development team

\(^{50}\) **PTR**: Problem Tracking Record, is a specific number used within Amadeus that relates to a problem that was experienced in an application
8.2. BPEL Monitoring Web UI and SoapWUI

Although they serve different purposes, both BPEL monitoring Web UI and the SoapWUI were implemented using the same technologies. Both Web applications were developed using Adobe Flex framework (68). Flex is a free open source framework, based on the proprietary Adobe Flash, for building rich internet applications. Flex is a set of technologies which solely focus on the presentation tier of a multi-tier application.

8.2.1. Purpose

**SoapWUI**

The goal of this Web application is to provide a simple UI for displaying all the information related to a running SoapUI extension instance. It provides the following features:

- Dynamically connects to a chosen host running the SoapUI extension
- Retrieve all the SoapUI projects from the selected host
- Retrieve all test suites, test cases, assertions…for a selected project
- Remotely run selected test cases
- Display the test results

**BPEL Monitoring Web UI**

The goal of this Web application is to provide a simple UI that will allow the monitoring of deployed BPEL processes and process instances of Apache Ode BPEL engine. It makes use of Flex and the Process and Instance management APIs of Apache Ode. The application has the following features:

- See deployed BPEL processes
- See BPEL process instances that were created, along with their variables
8.2.2. Architecture

The SoapWUI Web applications exchange messages with the Soap extension Web Service using SOAP 1.1 messages. The Web application requests and process the data received by updating the different data providers of the UI elements, in this case the different SoapUI projects, test suites,... Soap messages are processed and displayed in a graph format, to ease the data visualization for the user.

Figure 58: SoapWUI architecture

The same principle applies to the BPEL Monitoring Web UI, with the difference that it targets Apache Ode’s management Web Services. These services provides a practical interface to the BPEL engine allowing to retrieve the deployed processes, the various instances along with its internal variables which can be really helpful for inspecting the process data flow.

Figure 59: BPEL Monitoring Web UI architecture

Soap 1.2 is not supported by Flex at the moment.
8.2.3. Implementation details

These UIs were developed using Flex. Flex applications are built using the combination of two languages: MXML and ActionScript 3.

MXML is an XML-based user interface markup language that provides UI components as well as components for accessing server-side components.

ActionScript 3 is a programming language based on ECMAScript(69), a standardized scripting language, which is used for processing data, handling the interactivity of rich internet applications...using Adobe Flash-based technologies. ActionScript 3 also supports the ECMAScript for XML (E4X) (70) extension that enables native XML support within the language.

In order to interact with server-side components, Flex provides multiple means such as HTTPService, RemoteObject, WebService and so on. The one I was interested in and that I used is the WebService component, which allows you to access Web Services as its name indicates. It currently supports SOAP 1.1, XML Schema 1.0 (versions 1999, 2000, and 2001), and WSDL 1.1 RPC-encoded, RPC-literal, and document-literal (bare and wrapped style parameters).

Each received messages is processed using the E4X functionalities that ActionScript 3 offers. Soap messages (in the case of SoapWUI) as well as BPEL variables (in BPEL Monitoring Web UI) are dynamically transformed and displayed into graphs. For representing the data as graphs, I used Flare (71), a library for creating interactive data visualizations from the UC Berkeley Visualization Lab. Because the data in both cases is XML-based, a tree representation would fit naturally as you can see in Figure 60.

Figure 60: Visual representation of an XML message using Flare

---

52 As you may have noticed, nodes having children are colored. The graph is collapsible, and I found it difficult to remember which node had children especially when dealing with large graphs. This simple trick allows you to differentiate leaves from their parent nodes.
In Figure 61, you can see on the left side the textual representation of a SOAP message, and on the right side a visual representation of the same message using another Flash-based library, RaVis (72), which offers roughly the same functionalities as Flare in terms of complex data representation.
In Figure 62 you can see the UI that will allow you to see the deployed processes on the left side, and the process instances on the right side. When double clicking the instance, one can view the different variable(s) with their value that the instance had to process.

As I said earlier, the BPEL monitoring Web application makes use of Ode’s Web Services. You can have a look at the different operations these services offer in Figure 63. The entry point is the listAllProcesses operation which returns deployed processes. From that point on, you can use all other operations for retrieving instances (listInstances), retrieve variables (getScopeInfo + getVariableInfo), and so on.
Chapter 9

9. Evaluation and conclusion

9.1. Evaluation

Evaluating the underlying idea of the tool is rather biased process. The idea and my approach were warmly welcomed by the team I was in at Amadeus, and to quote my former unit-manager “it is exactly what we want”. Even if there wasn’t any existing Shopping Dossier Web Services until the very end of the internship (and here I owe a big thanks to Gerry), I showed the team that the idea is worth digging and that it can help them automate testing and improve the overall quality of their components. One drawback according to them is that it uses Web Services, which are not present yet.

The solution brings flexibility thanks to the various languages, tools and frameworks. The goal in the very end is to bring skilled testers and non-developers a bit closer. Testers can take advantage of the underestimated power of SoapUI, and non-developers can benefit from a great client-side technology such as Adobe Flex.

But there is no free lunch. Such ecosystem requires vast amount of knowledge, and this can definitively be seen as a huge handicap. Just think of the BPEL language, people have hard times to get their hands on and exploit its potential. I will advocate the idea that a tester should be someone who is highly skilled and therefore invaluable for a company. This skilled person should feel comfortable with the involved languages and with the SOA paradigm.

Also, when using this kind of ecosystem, you should always have in mind the big picture; otherwise you tend to get lost among all these components and feel frustrated. The goals here are not only to allow tester to create, maintain, organize, run and store powerful tests, but also to provide the possibility for non-developers to closely participate to the testing processes.

Although the learning curve of a tool such as SoapUI may not seem steep at first glance, in order to truly master its full potential you should know SoapUI APIs, XPath, XQuery, Groovy…Once you do, you can accomplish basically anything you want. Creating components to be added in such system requires you to play with different languages at the same time (especially during system testing).

All the different functional requirements discussed in Chapter 6 were fulfilled by the different tools and prototypes.

Non-functional requirements were not really directly stated for this project. But implicitly the performances of a test framework should be acceptable otherwise it will discourage you from testing at all. The various tools used in the ecosystem are quite responsive, and uses latest technologies. One of my biggest concerns is the efficiency of XML processing. All the major components uses XML data or and interact with XML-based messages. Choosing a StAx based parser such as Apache AXIOM was not really a choice but more an indirect benefit. My choice was to use Axis2 because it allowed me to create a lightweight Web Service server that could be embedded in SoapUI. According to (73), Apache AXIOM does perform really well comparing to other XML parsers.

Security was not a non-functional requirement in this project so it was not really taken into account. This kind of ecosystem should stay inside the company’s intranet and should not be exposed to the outside world. You may expose sensitive data when running test cases, so this infrastructure should stay put within your LAN. Clearly, security should be added to the Web applications so you can control people having access to the tests and the processes.
Some light load tests were performed to check out the SoapUI extension behavior when receiving a lot of requests. They will be discussed in the following section.

9.2. Testing

Each pieces of functionalities was obviously tested (it would not make a lot of sense to dissert about testing without applying it, would it?). I will focus on the test made for the prototypes I have developed. Although I did not have tons of time to test every part in depth, I manage to build some stable components.

Testing this kind of infrastructure is no easy task, but as for the SoapUI extension, I manage to develop it in a way that the operations do not depend on one another to correctly execute. This approach results in business logics which are completely isolated from one another and are easier to test in isolation. Of course, in the case of the Web application, it’s different because it calls the operations in a specific order, for instance:

1. Get all the projects
2. Get all the test suites
3. Run the test suites

When these operations are part of a flow, then the execution of one method depends on the results of its predecessor. Imagine that you cannot retrieve the project names; you won’t be able to retrieve its test suites since the operation requires a correct project name.

The testing of the SoapUI extension can be divided in two parts: the code testing, using JUnit, and integration testing using SoapUI. The particularity of this extension is that it needs SoapUI to execute because it refers to existing SoapUI projects, so running test cases without having SoapUI started up is impossible. So some tests had to be embedded into the .jar file in order to be run.

The second particularity is the debugging of this extension. As a matter of fact, SoapUI run in its own Java Runtime Environment, which make direct debugging from a Java IDE such as Eclipse impossible. The only way to debug the SoapUI extension was to use the remote debugging feature that Eclipse offers.

Once the extension is loaded and started in SoapUI, it deploys a Web Service which can be tested with SoapUI. This service was injected faulty SOAP messages which results in some nice exceptions that were not handled correctly. Those messages were manually created with correct and perturbed data values, and sent to the service to check out its response. Generally the test I created contains a combination of the assertions that SoapUI provides, such as:

- SimpleContains: this simple assertion checks if a specific token is present in the response
- SimpleNotContains: the opposite of the above assertion
- SchemaCompliance: checks if the response is compliant with the WSDL definition
- NotSoapFault: checks if the response is not a SOAP fault
- XPathContains: will match a specific XPath expression and its expected results against the response
- XQueryContains: same as above, but with an XQuery expression
- ResponseSLA: checks if the response occurred in a certain amount of time
- Groovy: assertion performed by a Groovy script
- ...

All these assertions are thus combined together in a test case so to guarantee a certain level of quality.
I also make use of SoapUI load test tool for sending many requests to the service in order to see if it can take up the load. Although I have nothing to compare it to, since I have made only one implementation using Axis2, and I did all my work on a laptop, the results showed that even under some quite stress, the Web Service keeps behaving correctly.

I started up with 5 threads which simply send requests every second with a small random delay between the requests. Everything went pretty fine with this load test, obviously the memory consumption was a little high but all the requests got a correct response and no errors whatsoever.

Then I choose other parameters with a different strategy:

- **Number of threads:** 10 threads
- **Strategy:** burst
- **Delay between the bursts:** 5 seconds
- **Burst length:** 20 seconds
- **Total load test time:** 3 minutes

At some point of the test, errors kept showing, resulting in almost half erroneous requests as you can see in Figure 64 in the last column `err`. On 6935 fired requests, 3703 resulted in error. The error was a `java.net.BindingException` complaining about an Address already in use and a response which is empty. The main reason of this error is that the socket has not enough time to close during each request resulting in an address which is already in use.

![Figure 64: Load test results](image)

Interestingly, the errors only showed up after some time as you can see in Figure 65 in red dots. I believe that at some point, once the first error occurred, it resulted in a death spiral failing subsequent requests.
Once I modified the SoapUI settings so to handle the socket correctly, I re-run the same load test with the same parameters, and this time, everything went perfectly fine as you can see in the pictures below.
SoapUI also enables you to create specific flows, link test cases together, transfer the parts of one response to the request of the next test request. This enabled me to create and test some operation flows pretty rapidly and efficiently. The “clone test” feature consents to reuse the previous tests, aimed at testing operation in isolation, so in order to put them into the wanted sequence.

As for the Web Applications, I made most of the testing using FlexUnit (74), a unit testing framework for Flex and ActionScript 3.0. Although it is a handy framework, it is still not very mature, and contains some problems I experienced. For instance, I created a simple test which aims at testing if a particular object is null, to assert that it’s null. The framework, coming with a visual test runner as you can see below, does not recognize the assertion and does not display it correctly. I am still looking for the reason.
Most of the testing was made in conjunction with the powerful debugger that Flex Builder 3, the IDE for building Flex applications, provides. Luckily for me, the E4X syntax made some things easier when it comes to verify XML messages that were retrieved from the SoapUI extension and from Apache Ode’s Web Services. I focus the testing on the displaying of the data rather than XML message verification. It is more convenient to use SoapUI for that.

As you may have understood, methodically testing these pieces is neither easy nor trivial. It requires the mastering of several testing framework, FlexUnit, JUnit, XMLUnit as well as XML related languages; XPath, XQuery and maybe also Groovy since SoapUI embeds a Groovy engine. Two debuggers were needed, the one from Flex Builder and the one to remotely debug the SoapUI extension.

This process is coupled with my way of developing which is: code a little, test a little, code a little, test a little… and the usage of trace (for Flex) and System.out.println for Java, two commands for outputting to the system console.


9.3. Conclusion and future work

The main goal of the first part of this thesis was to introduce and discuss the particularity of testing Web Services which are a key technology to develop server-side software components. Because of their nature and the many specifications evolving around them, they are inherently difficult to test.

I introduced different concepts from the software testing area, such as test automation, and discuss why a good balance between automated and manual tests is preferable. I introduced in details some methods for measuring the quality of software. I then talked about technologies which are used to build server side components, such as Java RMI and Corba, and finally one of the latest paradigms: Web Services. I detailed a language for creating composite Web services, BPEL, and talked about the various constructs that make up the language (in Appendix A). I proceeded by presenting ways and methods for testing Web Services, composite Web Services and outlined the ongoing research in this field along with some available tools for testing Web Services.

The goal of the second part of the thesis was to develop a proof of concept which makes use of BPEL processes, that is, composite Web Services as primary artifacts for creating test scenarios. Those scenarios, when deployed, become Web Services. Those services are then exercised by a specific test tool, SoapUI. This tool was then extended to provide some of its functionalities to Web applications.

To summarize, the contributions in this thesis are:

- A SoapUI extension, embedding a Axis2 Web Service engine, which goal is to provide to a Web application an interface to connect to a running SoapUI instance
- A useful Groovy script which exports the data results for reporting purposes
- An algorithm for transforming XML messages into a graph
- A Web user interface for monitoring deployed processes and process instances
- A Web user interface for connecting to a node which runs the SoapUI extension I have created.

The extension that I have built for SoapUI allows testers to create and deploy a Web Service which enable remote Web Applications to connect to SoapUI in order to list all the projects, test suites, test cases, assertions…of the tester, view the request in a friendly way and to run the test cases and gather the results. The goal of this extension was not provide all the capabilities of SoapUI in a remote fashion but rather focuses on the most important parts such as WSDL test requests and responses, and to enable testers to share their current tests with non-testers.

I have also developed an Apache Ode management user interface prototype which enables testers to monitor the Apache Ode BPEL engine. This engine comes with some process management and process instances management interfaces. The user interface, although partial, reflects the importance of monitoring the processes themselves, after all they are our interface to our SUT. That is why future work should focus on verifying the BPEL process itself, by making sure that all the various paths, conditions…it offers are all traversed and tested. This can only be done by an in-depth analysis of the process and data flows. One can also think of deriving the minimal data needed for reaching 100% path coverage of the process, thus minimizing tests cases. Deriving automatically test data from the BPEL process is pretty difficult due to the complexity of XML Schema itself, the various constraints on the data which are to be solved…In order to do so, one has first to settle down on a formal representation of XML Schema, and derive data from it.

The user interface which connects to SoapUI can also be improved in order to provide further functionalities. This user interface should evolve according to the functionalities the SoapUI extension provides. Making this Web application user friendly is challenging, it should present a subset of what the tester does to a less skilled person, and it should display complex data such as SOAP messages in
an interactive and friendly way, using graphs for instance. Making use of advanced data visualization libraries such as Flare is thus required.

The implemented parts are prototypes serving only as proof of concepts; they were not developed to go into production. They are an on-going work which will continue in order to have fully-fledged applications. Therefore the current implementation is not complete at all level and most probably contains defects. They were built so in order to promote the idea of using test scenarios which can be built as BPEL processes, deployed as Web Services and tested using tools such as SoapUI.

From this thesis, it can be concluded that testing is critical to develop quality applications, that testing methods from the past can be adapted to new paradigms such as Web Services. These paradigms force us to think and build flexible tools which can adapt to this new paradigm, and eventually help us chase defects in the software.

Testing is an amazingly interesting field of software engineering. It gathers together concepts from different areas with the intent of improving our skills box. The more we fill in this box with meaningful methods, the better we will get at building quality software.

9.3.1. Suggestions for future work and enhancements

As I said earlier further work is needed that encompasses the different aspects of the prototype. One of the major future works should concentrate on automating the testing of BPEL processes, as well as test case derivation from a process. This could alleviate substantially the pain of thinking about the test data for each process. Data can be derived from the XML Schema, and equivalence classes could be created automatically. Test cases could go and take the different values from a data pool during execution.

Another major concern is the correctness of the process itself. Due to its complexity, BPEL processes may fail to operate in various ways, especially if the processes are very large and intricate. Model checking can help in verifying BPEL processes, but again, this requires major efforts since BPEL is a semi-formal language. Applying formal verification techniques requires a formal representation of BPEL. Lots of researchers focus on this particular topic.

An additional improvement can be made at the SoapUI extension level. Instead of having a ‘lazy’ Web Service that sits there waiting for request, we could, in the future, develop a push-based Web Service in order to keep the data between SoapWUI and SoapUI in sync. This would require to have access to implementation of several WS-* specification such as WS-BaseNotification, WS-Eventing … as well as a support of the same specifications from the client side.

The SoapUI test runner can also be improved in a way that it would have the possibility to exchange tasks with other instances as well as providing a task migration mechanism at execution time if one machine is overloaded, or simply to distribute tests fairly across nodes. This network of SoapUI instances could share, execute tests dynamically… This would require either the complete rethinking of the test runner or a separate extension which enable distributed testing.

53 Understand tests.
This engine for distributing the tasks between SoapUI instances could be implemented using a language which fits the job such as Erlang, or could use a lightweight grid infrastructure such as GridGain (75) for instance.

From the SoapWUI point of view, you can easily modify it will so to access several SoapUI extensions at the same time. In this way, you could also group same projects which are located on different nodes and display it in a convenient way to the user.

The BPEL monitoring Web UI should be completed by implementing maybe all the functionalities that Apache Ode’s Web Services provide. I personally consider it to be a project *per se* because of the many capabilities the interface provides.

![Figure 69: Future improvement: dynamic test migration and distributed execution](image)
Bibliography


Appendix A

A.1 Notorious software failures

- Ariane 5 Flight 501 explosion

The infamous European rocket Ariane 5 disintegrated in less than 40 seconds (at H0 +39 seconds to be accurate) after its launch on June the 4th 1996, resulting in an explosion worth hundreds of US$ millions and years of researches thrown away.

The cause: *an arithmetic overflow*;

**Brief explanation:**

Ariane 5 reused Ariane 4 specifications, and there were nothing wrong with that since everything went perfectly well with Ariane 4’s previous launches. But engineers, due to timing and budget constraints, did not thoroughly execute specific regression tests, defined by (6) as the ‘selective retesting of a system or component to verify that modifications have not caused unintended effects’. The specific component to re-test was the inertial reference guidance computer software which checks the flight path and re-corrects it in case of a path deviation. Some sequences of operations were computed after the launch for 40 seconds, which embedded a faulty conversion from a 64-bit floating point value (used by Ariane 5’s software) to a 16-bit signed integer (used by Ariane 4’s software) value, setting off a chain of uncaught errors. Those errors mislead the launcher’s main computer into thinking that the flight path was incorrect, naturally triggering a path rectification that was unnecessary. The rapid change in path tore apart the boosters’ nozzles, resulting in one of the most expensive (in terms of money) software failure in the history. More precise details can be found in (76).

- Mars Exploration Rover A (known as *Spirit*)

NASA prepared two robots, Spirit and Opportunity, aimed at exploring the red planet, Mars. One of rover, Spirit, failed to respond to the central command office during several days after its landing on Mars.

The cause: *bad memory allocation*;

**Brief explanation:**

Spirit was keeping rebooting itself over and over again. NASA scientists eventually discovered that it was due to a bad memory allocation to a nonexistent address prompting to a fatal error that led the robot into a reboot cycle. They finally disabled the dynamic memory allocation that caused the flaw and recovered Spirit which was able to continue its mission.
A.2 The minefield paradox

Consider the minefield as the system you want to test, the red shapes as the defects and the dashed arrows as tests you have executed.

In Figure 71, the blue circles represent discovered mines, i.e. bugs in our software that we are going to fix. Imagine now that after some changes you have made to your system, you repeat those tests, what will they tell you? Nothing new, because totally repeatable tests do not bring any added value as stated in (9).

In order to overcome this issue, you might prospect variable tests as pictured in Figure 72.
I pointed out this analogy because it is one of the many pitfalls of regression testing, and therefore is worth mentioning that you cannot rely solely on regression tests.
Appendix B

B.1 WSDL 2.0 example

Below, a small WSDL 2.0 example, so you can compare it to the old one. As you might notice, some XML elements are different in names and new attributes can be found, such as the MEP to use for an operation. But there are a lot more differences; details are available in (77),(78) and (79).

```xml
<description targetNamespace="http://great.example.com/2004/services/reservationList">

This document describes the GreatH Reservation List Web services. Use this service to retrieve lists of reservations based on a variety of search criteria.

</documentation>

<types>
  <xs:import namespace="http://great.example.com/2004/schemas/reservationDetails" schemaLocation="reservationDetails.xsd"/>
</types>

<interface name="reservationListInterface">
  <operation name="retrieve" pattern="http://www.w3.org/2001/XMLSchema-instance">
    <input messageLabel="In" element="Innone"/>
    <output messageLabel="Out" element="listReservationList"/>
  </operation>

  <operation name="RetrieveByConfirmationNumber" pattern="http://www.w3.org/2001/XMLSchema-instance">
    <input messageLabel="In" element="detailsConfirmationNumber"/>
    <output messageLabel="Out" element="listReservationList"/>
  </operation>

  <operation name="RetrieveByCheckInDate" pattern="http://www.w3.org/2001/XMLSchema-instance">
    <input messageLabel="In" element="detailsCheckInDate"/>
    <output messageLabel="Out" element="listReservationList"/>
  </operation>

  <operation name="RetrieveByCheckOutDate" pattern="http://www.w3.org/2001/XMLSchema-instance">
    <input messageLabel="In" element="detailsCheckOutDate"/>
    <output messageLabel="Out" element="listReservationList"/>
  </operation>

</interface>
</description>
```

Figure 73: Example of a WSDL 2.0 – ReservationList Web service

In this Web service example from (79); you can see that it imports two XML schema files which define the type of data in its messages it receives and/or sends. The two schemas are shown below.

You can also see the four different operations it exposes:

- **Retrieve**, which returns a list of reservations, in the form of a list, `reservationList`, of complex XML elements of type `reservation`.

- **RetrieveByConfirmationNumber**, receives a string as input parameter, `confirmationNumber` and returns `reservationList` as the previous operation.
- **RetrieveByCheckInDate**, receives a date as input parameter, `checkInDate`, and returns a `reservationList`.

- **RetrieveByCheckOutDate**, receives a date as input parameter, `checkOutDate`, and returns a `reservationList`.

You can see in Figure 74 and in Figure 75 the data type of the elements of the messages that are expected when interacting with the `ReservationList` Web service.

```xml
<schema

elementFormDefault="qualified"
targetNamespace="http://greath.example.com/2004/schemas/reservationList"/>
<import

namespace="http://www.w3.org/2001/XMLSchema-instance"/>
<import

namespace="http://greath.example.com/2004/schemas/reservationDetails" schemaLocation="reservationDetails.xsd"/>
<element name="reservation">
  <annotation>
    A reservation contains the confirmation number, check-in and check-out dates, and a reference to a Reservation Details Web service.
  </annotation>
  <complexType>
    <sequence>
      <element ref="details:confirmationNumber"/>
      <element ref="details:checkInDate"/>
      <element ref="details:checkOutDate"/>
      <element ref="details:reservationDetailsSOAEndpoint"/>
    </sequence>
  </complexType>
</element>
<element name="reservationList">
  <annotation>
    A reservation list contains a sequence of zero or more reservations.
  </annotation>
  <complexType>
    <sequence>
      <element ref="tns:reservation" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute ref="wsdl:wsdlLocation"/>
  </complexType>
</element>
</schema>
```

Figure 74: The `ReservationList XML schema`
Figure 75: The ReservationDetails XML schema
B.2 SOAP encoding styles

Suppose we have the following Java method called \texttt{myMethod}, exposed as a Web Service, which takes two parameters, \( x \) of type \texttt{int} and \( y \) of type \texttt{float}.

\begin{verbatim}
public void myMethod(int x, float y);
\end{verbatim}

Let us see how the different styles present the information in the WSDL interface and how the SOAP message related to the interface is represented, and let us see their pros and cons.

When you use the RPC style, you have to conform to the method structure you are calling, that is; the SOAP body structure in terms of the method name, parameters...has to be the same as the remote method structure you are calling.

- \textit{RPC-encoded}

The WSDL interface part for the method will look like this:

\begin{verbatim}
<message name="myMethodRequest">
  <part name="x" type="xsd:int"/>
  <part name="y" type="xsd:float"/>
</message>

<portType name="PT">
  <operation name="myMethod">
    <input message="myMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>

<binding .../>
\end{verbatim}

The SOAP message that will be sent by a remote client will look like this:

\begin{verbatim}
<soap:envelope>
  <soap:body>
    <myMethod>
      <x xsi:type="xsd:int">5</x>
      <y xsi:type="xsd:float">5.0</y>
    </myMethod>
  </soap:body>
</soap:envelope>
\end{verbatim}

One benefit, as you can see, is that the WSDL interface (I just took the part I was interested in) is pretty straightforward. Another benefit is that the method name appears in the message which is pretty helpful when you have to dispatch the message to the actual implementation of the operation.

On the other hand validation of the SOAP message is not an easy task since only the \(<x...>5</x>\) and \(<y...>5.0</y>\) contain things defined in an XML Schema (\texttt{xsi:type="xsd:int"} and \texttt{xsi:type="xsd:float"}). The rest of the message cannot be easily verified. Also, this type of encoding info (\texttt{xsi:type="xsd:int"}...) adds a certain overhead and degrades throughput performance since you
have to pass the encoding information each time you send a SOAP message. This style is not compliant with the WS-I\textsuperscript{54} Basic Profile\textsuperscript{55}.

- **RPC-literal**

The WSDL interface part for the method will look like this:

```xml
<message name="myMethodRequest">
  <part name="x" type="xsd:int"/>
  <part name="y" type="xsd:float"/>
</message>
:message name="empty">

<portType name="PT">
  <operation name="myMethod">
    <input message="myMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>

The SOAP message that will be sent by a remote client will look like this:

```xml
<soap:envelope>
  <soap:body>
    <myMethod>
      <x>5</x>
      <y>5.0</y>
    </myMethod>
  </soap:body>
</soap:envelope>
```

The WSDL interface is still pretty straightforward and readable. The SOAP message still contains the method name. The encoding information for the parameters in the SOAP message is eliminated. Another advantage is that this style is WS-I compliant.

The big drawback is when you want to validate the message, only \( x \) and \( y \) type information related to a schema and can be validate. The rest of the message is hard to validate.

\textsuperscript{54} The Web Services Interoperability organization (WS-I) is an open industry which promotes best practices to adopt in order to maximize Web Services interoperability. They deliver several profiles for security, interoperability…

\textsuperscript{55} The WS-I Basic Profile is an aggregation of rules to adopt when developing Web Services-based application in order to improve interoperability
- When you use Document style, you are free to structure the content of the body in any way you like.

  - **Document-encoded**
    This style is not WS-I compliant and will not be detailed.

  - **Document-literal**

The WSDL interface part for the method will look like this:

```xml
<types>
  <schema>
    <element name="xElement" type="xsd:int"/>
    <element name="yElement" type="xsd:float"/>
  </schema>
</types>

<message name="myMethodRequest">
  <part name="x" element="xElement"/>
  <part name="y" element="yElement"/>
</message>

<message name="empty"/>

<portType name="PT">
  <operation name="myMethod">
    <input message="myMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>

<binding .../>
```

The SOAP message that will be sent by a remote client will look like this:

```xml
<soap:envelope>
  <soap:body>
    <xElement>5</xElement>
    <yElement>5.0</yElement>
  </soap:body>
</soap:envelope>
```

One big advantage is that this time you can validate the content of the SOAP message because every element present in the SOAP Body is defined in the schema part of the WSDL interface. There is no encoding information in the SOAP message, and this style is WS-I compliant but with some restrictions due to its weaknesses.

One weakness is that the WSDL interface starts to be complex in its structure and hard to read (even though it was not meant to be read by a human in the first place). Another disadvantage is that the operation name is lost in the SOAP message, and it can be hard to dispatch it (sometimes impossible). The WS-I Basic profile allows only one child for the SOAP Body element, and as you can see, the SOAP Body has two children: `xElement` and `yElement`. 
- **Document-literal wrapped**

This style is a special one which derived from the previous one, but was rearranged so in order to recover the method name in the SOAP message.

The WSDL interface part for the method will look like this:

```xml
<types>
  <schema>
    <element name="myMethod">
      <complexType>
        <sequence>
          <element name="x" type="xsd:int"/>
          <element name="y" type="xsd:float"/>
        </sequence>
      </complexType>
    </element>
    <element name="myMethodResponse">
      <complexType/>
    </element>
  </schema>
</types>
<message name="myMethodRequest">
  <part name="parameters" element="myMethod"/>
</message>
<message name="empty">
  <part name="parameters" element="myMethodResponse"/>
</message>
<portType name="PT">
  <operation name="myMethod">
    <input message="myMethodRequest"/>
    <output message="empty"/>
  </operation>
</portType>
<binding .../>
```

The SOAP message that will be sent by a remote client will look like this:

```xml
<soap:envelope>
  <soap:body>
    <myMethod>
      <x>5</x>
      <y>5.0</y>
    </myMethod>
  </soap:body>
</soap:envelope>
```

It is pretty obvious that the major drawback is the fact that the WSDL interface is more complex than all the previous styles.

You can see that the SOAP message looks similar to the RPC-literal, but in reality they are not. In the SOAP RPC-literal message, the `<myMethod>` element is the child of the SOAP Body element, and has the same name as the operation. In Document-literal wrapped SOAP message, the `<myMethod>`
clause is the name of the wrapper element which the input message’s part refers to. This is a clever way of putting back the method name in the SOAP message.
This particular style group the previous benefits of the Document-literal style, that is, the type encoding information is not present, everything that appears in the SOAP Body is defined in the schema and can be validate, and the method name is present in the message.

This style is WS-I compliant, and meets the restrictions which specify that the SOAP Body has to have only one child.

Which style to use?
Well, here I am going to quote one of my teacher’s favorite responses:”It depends”. Clearly, you will choose one WS-I compliant style, because interoperability does matter. They all have their own advantages and drawbacks and they all fit specific contexts.

For instance, if message’s validation is one of your top concerns, obviously you will prefer the Document-literal style because the content is defined in an XML schema and you can use any XML validator which is widely available to validate the message.

If you need to route your message to the server-side method’s actual implementation, because pure Document literal non Wrapped does not show the method name, you will prefer the RPC-literal style.

One drawback of literal style is when facing cyclic references (for instance data graphs) in the content of a message, which will results in a larger message to transmit because of the repetition of the elements, therefore you can use the RPC-encoded style (even if it is not WS-I compliant).

More precise details about the SOAP protocols can be found in (80),(81) and (82).
B.3 Detailed explanation of the BPEL language constructs

A BPEL process is a container of other declared activities such as relationships to other Web Services, data variables, and other executable activities.

On the outermost side of a BPEL process, we can see the process element.

```
<bpws:process exitOnStandardFault="yes" name="purchaseOrderProcess"
    targetNamespace="http://example.com/ws-bp/purchase"
    xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:lns="http://manufacturing.org/wsdl/purchase">

  <bpws:variables>
    <bpws:variable messageType="lns:POMessage" name="PO" />
    <bpws:variable messageType="lns:InvMessage" name="Invoice" />
    <bpws:variable messageType="lns:shippingRequestMessage" name="shippingRequest" />
    <bpws:variable messageType="lns:shippingInfoMessage" name="shippingInfo" />
    <bpws:variable messageType="lns:scheduleMessage" name="shippingSchedule" />
  </bpws:variables>

  <bpws:partnerLinks>
    <bpws:partnerLink myRole="purchaseService" name="purchasing"
        partnerLinkType="lns:purchasingLT" />
    <bpws:partnerLink myRole="invoiceRequester" name="invoicing"
        partnerLinkType="lns:invoicingLT" partnerRole="invoiceService" />
    <bpws:partnerLink myRole="shippingRequester" name="shipping"
        partnerLinkType="lns:shippingLT" partnerRole="shippingService" />
    <bpws:partnerLink name="scheduling"
        partnerLinkType="lns:schedulingLT" partnerRole="schedulingService" />
  </bpws:partnerLinks>

</bpws:process>
```

A process element has the following attributes defining:

- its own name: name
- a targetNamespace
- additional namespace for Abstract Processes

A BPEL process generally involves long-running transactions, therefore representing a stateful service, that is, it retains values throughout its execution. BPEL uses the blackboard approach, where variables are shared amongst activities within the same scope.

A variable element includes the following attributes:

- a name
- a type. There are three sorts of type:
  - messageType
  - type
  - element

A BPEL variable can come from either a WSDL message type, XML schema (simple and complex type) or XML schema elements.

In BPEL, the interaction partners are referenced using the partnerLink activity as you can see in Figure 25 and the XML version just below.

Each partnerLink has attributes for:

- Defining its name: name
- Defining its type: partnerLinkType
- Defining its own role played during the process: myRole
- **Defining its role as a partner:** *partnerRole*

*Receive* is a simple activity which purpose is to collect a message from an external partner.

```
<bpws:receive createInstance="yes" operation="sendPurchaseOrder"
    partnerLink="purchasing" portType="lns:purchaseOrderPT"
    variable="PO">
    <bpws:documentation>
    Receive Purchase Order
    </bpws:documentation>
</bpws:receive>
```

Since it always specify a partner, it has to provide the name of the *partnerLink*, in the example above it provides the name of the partnerLink that the process itself plays along with an *operation* that it provides to the external world. The data it receives is stored into a *variable* that may be used later in the process.

The attribute *createInstance* specifies, when set to “yes”, that when the process receives a new message, it instantiates a new BPEL process instance. When set to “no”, the message will be consumed by a running instance.

The *reply* activity is used in conjunction with the *receive* activity, so in order to send a message back to the process initiator. This *receive-reply* conjunction forms a request-response message exchange pattern which is defined as an operation in the WSDL interface. As you can see below, the attributes are nearly the same as the *receive* activity above.

```
<bpws:reply operation="sendPurchaseOrder"
    partnerLink="purchasing" portType="lns:purchaseOrderPT"
    variable="Invoice">
    <bpws:documentation>Invoice Processing</bpws:documentation>
</bpws:reply>
```
In order to communicate with external Web Services, a BPEL process uses the *invoke activity*.

```xml
<bpws:invoke inputVariable="PO" name="Invoke-production_scheduling"
    operation="requestProductionScheduling" partnerLink="scheduling"
    portType="lns:schedulingPT">
    <bpws:documentation>
        Initiate Production Scheduling
    </bpws:documentation>
</bpws:invoke>
```

The *invoke* activity is used to make a call to an external Web Service provided by one partnerLink. Its attributes are:

- name of the invocation: *name*
- name of the operation to call on the external Web Service: *operation*
- name of the variable to send to the external Web Service as an input: *inputVariable*
- name of the variable that will receive the response from external Web Service: *outputVariable*
- partner name to invoke: *partnerLink*
- port type of the partnerLink: *portType*

The input and output variables may be optional depending if the message exchange pattern is a request-response or a one-way operation.

The *scope* activity is a container that provides a literal context. This context modifies the behavior of its enclosed activities. For instance, a variable defined within a scope cannot be accessed from outside the scope. It provides *isolation* mechanisms.

```xml
<scope>
    <faultHandlers>...</faultHandlers>
    <flow>
        <invoke partnerLink="Seller"
            portType="Sell:Purchasing"
            operation="Purchase"
            inputVariable="sendPO"
            outputVariable="getResponse" />
        <invoke partnerLink="Shipper"
            portType="Ship:TransportOrders"
            operation="OrderShipment"
            inputVariable="sendShipOrder"
            outputVariable="shipAck" />
    </flow>
</scope>
```
The *sequence* activity contains one or more enclosed activities that are performed sequentially.

```xml
<sequence>
  <flow>...</flow>
  <scope>...</scope>
  <pick>...</pick>
</sequence>
```

It is only when the last enclosed activity has completed that the *sequence* activity is complete.

The *flow* activity provides mechanisms for parallel processing. In the example below, the two invocations are started concurrently, and once they finish, the *transferMoney* invocation is executed.

```xml
<sequence>
  <flow>
    <invoke partnerLink="Seller" ... />  
    <invoke partnerLink="Shipper" ... />  
  </flow>
  <invoke partnerLink="Bank" name="transferMoney" ... />  
</sequence>
```

A flow completes only when all its enclosed activities have completed. In Figure 27, three sequences are started concurrently.

The *link* activity is used for syncing activities together with the use of collections of sources and targets. This introduces a dependency mechanism that can be useful when some kind of causal order has to be mimicked.

```xml
<bpws:links>
  <bpws:link name="ship-to-invoice" />
  <bpws:link name="ship-to-scheduling" />
</bpws:links>
```

A *target* is an activity that depends on another one, the *source*. For instance, in the initial example in Figure 28, the *shippingRequest* invoke activity (represented by the ‘decide on shipper’ UML activity) is the source of a target activity, that is, the *shippingInfo* invoke activity (represented by the ‘Complete Price Calculation’ UML activity). This is visually pictured using an activity diagram just below with UML comments.
Special conditions can be attached to the source and/or the target, that is, the transitionCondition and the joinCondition.

A transitionCondition is a transition condition associated with a link that represents a guard that has to be evaluated prior to traversing the link. If no transitionCondition is specified, then the link’s status is true. This offers a mechanism to split the control flow based on conditions.

A joinCondition is a XPath boolean expression used to merge split control flow. For instance, a joinCondition can be used to enforce an activity to wait for all its links to be true before executing itself.

As you noticed, links provides a practical synchronization mechanism. But this can lead to a dead end if not used correctly. In order to avoid dead ends, an approach called Dead Path Elimination (DPE) provides two mechanisms for dealing with false join conditions:

- either you used a fault handler to cope with a joinFailure error that is thrown when facing false join conditions
- or you can make use of the suppressJoinFailure attribute in the appropriate activities so the false link status will be propagated
BPEL offers exception handling capabilities with `faultHandlers`. A fault handler is attached to a `scope`, an `invoke` or a `process` as pictured in Figure 27. When an `orderFaultType` message is received, the process catches it, and sends a specific reply to the customer saying that something went wrong and that it cannot complete the order.

```xml
<bpws:catch faultMessageType="lns:orderFaultType"
faultName="lns:cannotCompleteOrder" faultVariable="POFault">
  <bpws:reply faultName="bpws:cannotCompleteOrder"
    operation="sendPurchaseOrder" partnerLink="purchasing"
    portType="lns:purchaseOrderPT" variable="POFault"/>
</bpws:catch>
</bpws:faultHandlers>
```

A `faultHandler` can have either one or multiple `catch` constructs. Each of them must provide an activity (such as `reply`) that deals with the caught error.

A `faultHandler` can also have one instance of a `catchAll` construct which acts as a generalized fault handling mechanism. If you happened to deal with multiple exceptions and you don’t want to make any distinctions between them, `catchAll` will help you responding to the caught exceptions the same way.

BPEL also provides compensation mechanisms through the use of a `compensationHandler`. Long-running transactions can create persistent effects. This mechanism helps you to undo the effects when needed. A `compensationHandler` can be called from a `compensateScope` or a `compensate` activity, which may be enclosed into a fault handler, compensation handler or even a termination handler of an immediately enclosing scope.

```xml
<scope name="S1">
  <faultHandlers>
    <catchAll>
      <compensateScope target="S2"/>
    </catchAll>
  </faultHandlers>
  <sequence>
    <scope name="S2">
      <compensationHandler>
        <!-- undo work -->
      </compensationHandler>
      <!-- do some work -->
    </scope>
    <!-- do more work -->
    <!-- a fault is thrown here; results of S2 must be undone -->
  </sequence>
</scope>
```
As you saw earlier, you can decide to create new BPEL process instance each time the process receives a new message (setting the createInstance attribute of a receive activity to “yes”). On the other hand, sometimes a message reception does not lead to the creation of a new BPEL process instance, and therefore the message has to find the running process instance it is designed for. BPEL provides correlation to answer this need. The approach is the following:

In our purchase order process example, the customer already has a piece of data that uniquely identifies him: its customer identifier. What if the same customer sends multiple orders? Well in that case one can use an order number for each order. All those piece of data are normally part of the message request that has been received. BPEL allows you, through the use of properties, to define parts of message that can be represented as pieces of correlation information. Using property alias, a process can defines the location of these properties in the message.

On the WSDL part, you define the elements to be considered as properties. On the BPEL’s side, you make use of the correlationSet activity to access the properties. This enables you to uniquely identify a process instance for a specific purchase order using the customerID and the orderNumber.
You can in the above code block the WSDL interface defining the *properties* and *properties aliases*. In the code block below, the BPEL process makes use of these correlated data using the *correlationSet* activity.

```xml
<process name="purchaseOrderProcess" ...
  <correlationSets>
    <correlationSet name="PurchaseOrder" properties="cor:customerId cor:orderNumber" />
  </correlationSets>
  ...
  <onEvent partnerLink="purchasing" operation="queryPurchaseOrderStatus" ...
    <correlations>
      <correlation set="PurchaseOrder" initiate="no" />
    </correlations>
  </onEvent>
  <onEvent partnerLink="purchasing" operation="cancelPurchaseOrder" ...
    <correlations>
      <correlation set="PurchaseOrder" initiate="no" />
    </correlations>
  </onEvent>
  ...
  <sequence>
    <receive partnerLink="purchasing" operation="sendPurchaseOrder" ..., createInstance="yes">
      ...
    </receive>
    ...
    <reply partnerLink="purchasing" operation="sendPurchaseOrder" ...
      <correlations>
        <correlation set="PurchaseOrder" initiate="yes" />
      </correlations>
    </reply>
    ...
  </sequence>
  ...
</process>
```
Suppose now the following scenario:

A process has two *receive* activities that create an instance of process. Sometimes you just don’t want to create two instances upon reception of message because the messages have to be processed by the same instance. BPEL offers you mechanisms to deal with multiple events processing with correlations.

The example above waits for two messages to be received: one from a buyer (*receiveSellerInformation*) and the other from a seller (*receiveBuyerInformation*). It does not matter the order in which the messages are received, only the first received message will create an instance of the process. When the other message will be received, it will not create a new instance, and the running instance will consume the message. It is only when the two messages are received that the process proceeds to its next execution step because of the *joinCondition* activity.
All the former examples used blocking primitives, that is, the execution of the process is blocked until a message is received for instance. Imagine now a purchase order that has been instantiated and currently running has to be cancelled by the buyer. BPEL allows you to define concurrent events to be received in parallel to the normal execution using eventHandlers and onEvent activities.

```xml
<process name="purchaseOrderProcess" ...
<eventHandlers>
  <onEvent partnerLink="purchasing" operation="queryOrderStatus" ...>
    <scope>...</scope>
  </onEvent>
  <onEvent partnerLink="purchasing" operation="cancelOrder" ...>
    <scope>...</scope>
  </onEvent>
</eventHandlers>
```

A special handler is used to force the termination of the process being executed. When an error is caught and processed by the faultHandler, all other activities within the related scope are terminated. Some others are allowed to continue their processing, such as assign, throw, rethrow, exit and empty.

```xml
<scope>
  <terminationHandler>
    <!-- clean up resources in case of forced termination -->
  </terminationHandler>
  <sequence>
    <!-- do work -->
  </sequence>
</scope>
```
The *pick* activity waits for a specific event and, upon reception, executes the activity associated with that event. The event it receives is one of the following two events:

- **onAlarm**: corresponds to a timer-based alarm.
- **onMessage**: same principle as the *receive* activity.

```xml
<pick>
  <onMessage partnerLink="buyer" operation="inputLineItem" variable="lineItem">
    <!-- activity to add line item to order -->
  </onMessage>

  <onMessage partnerLink="buyer" operation="orderComplete" variable="completionDetail">
    <!-- activity to perform order completion -->
  </onMessage>

  <onAlarm>
    <for>'P3DT10H'</for>
    <!-- handle timeout for order completion -->
  </onAlarm>
</pick>
```

In the above example, if after 3 days and 10 hours the process did not receive one of the two messages it is expecting, then the attached logic to the timer will be processed.

Other simpler activities include:

<table>
<thead>
<tr>
<th>Activities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>used to update variables</td>
</tr>
<tr>
<td>empty</td>
<td>do nothing</td>
</tr>
<tr>
<td>exit</td>
<td>stop the process instance immediately</td>
</tr>
<tr>
<td>if-else</td>
<td>provides conditional construct</td>
</tr>
<tr>
<td>rethrow</td>
<td>used within a <em>faultHandler</em> to propagate caught errors</td>
</tr>
<tr>
<td>throw</td>
<td>used to signal explicitly internal faults</td>
</tr>
<tr>
<td>validate</td>
<td>used to validate messages or variables against a WSDL message type or an XML schema type or element.</td>
</tr>
<tr>
<td>wait</td>
<td>provides delay mechanisms</td>
</tr>
<tr>
<td>**while,**repeatUntil,<strong>forEach</strong></td>
<td>provides loop mechanisms</td>
</tr>
</tbody>
</table>

The BPEL language supports extensibility mechanisms. For instance, you can use other expression language than XPath 1.0 (for example Apache ODE\(^{56}\) engine supports XPath 2.0 expressions). Much more details regarding the constructs can be found in (17).

\(^{56}\) Apache Orchestration Director Engine (ODE) is an open source BPEL engine that executes WS-BPEL 2.0 processes. It also supports the older version of BPEL, which is BPEL for Web Services (BPEL4WS).