7 Web Application Testing

Giuseppe A. Di Lucca, Anna Rita Fasolino

Abstract: Web applications are characterised by peculiarities that differentiate them from any other software application. These peculiarities affect their testing in several ways, which may result in harder than traditional application testing. Suitable methods and techniques have to be defined and used to test Web applications effectively. This chapter will present the main differences between Web applications and traditional ones, and how these differences impact the testing of Web applications. It also discusses relevant contributions in the field of Web application testing, proposed recently. The focus of the chapter is mainly on testing the functionality of a Web application, although discussions about the testing of non-functional requirements are provided too. Readers are required to have a general knowledge of software testing and Web technologies.

Keywords: Web engineering, Web application testing, Software testing.

7.1 Introduction

In the last decade, with the wide diffusion of the Internet, a growing market request for Web sites and applications has been recorded. As more and more organisations exploit the World Wide Web (WWW) to offer their services and to be reached by larger numbers of customers and users, the request for high-quality Web applications satisfying security, scalability, reliability, and accessibility requirements has grown steadily. In such a scenario, testing Web applications to verify their quality became a crucial problem.

Unfortunately, due to market pressure and very short time-to-market, the testing of Web applications is often neglected by developers, as it is considered to be time-consuming and lack a significant payoff [11]. An inversion of this trend may be obtained if testing models, methods, techniques, and tools that allow testing processes to be carried out effectively and in a cost-effective manner are available.

Although Web application testing shares similar objectives to those of "traditional" application testing, there are some key differences between testing a traditional software system and testing a Web application: the specific features exhibited by Web applications, and not included in other software systems, must be considered to comprehend these differences.

A Web application can be considered as a distributed system, with a client–server or multi-tier architecture, including the following characteristics:

- It can be accessed concurrently by a *wide number of users* distributed all over in the world.
- It runs on complex, *heterogeneous execution environments*, composed of different hardware, network connections, operating systems, Web servers, and Web browsers.
- It has an extremely *heterogeneous* nature that depends on the large variety of software components that it usually includes. These components can be built by different technologies (i.e. different programming languages and models), and can be of a different nature (i.e. new components generated from scratch, legacy ones, hypermedia components, COTS, etc.).
- It is able to generate *software components at run time* according to user inputs and server status.

Each aspect described in the previous list produces new testing challenges and perspectives. As an example, effective solutions need to be identified for executing performance and availability testing to verify a Web application's behaviour when accessed concurrently by a large number of users. Moreover, as users may utilise browsers with different Web content rendering capabilities, Web applications must be tested to make sure that the expected application's behaviour using different Web browsers, operating systems, and middleware is the one expected. Another critical feature of a Web application to be specifically tested is its security and ability to be protected from unauthorised access. The different technologies used to implement Web application components influence the complexity and cost of setting up a testing environment required to test each component. In addition, the different mechanisms used to integrate distributed components produce various levels of coupling and inter-component data flow, impacting the cost for being tested effectively. As for the existence of dynamically generated software components, the issue here is to cope with the difficulty of generating and rerunning the same conditions that produced each component.

Finally, Web application testing also needs to take into account failures in the application's required services/functionality, to verify the conformance of the application's behaviour to specified functional requirements. Considering that the components of a Web application are usually accessed by navigation mechanisms implemented by hyper-textual links, a specific verification activity also needs to be devised to check link integrity, to assure that no unreachable components or pending/broken links are included in the application.

Problems and questions regarding Web applications' testing are, therefore, numerous and complex. In this chapter we discuss these problems and questions and present possible solutions, proposed by researchers, from both academic and industrial settings.

We use two separate perspectives to analyse Web application testing: the first considers aspects related to testing the non-functional requirements of a Web application; the second considers the issue of testing the functionality offered by Web applications.

Section 7.2 introduces several types of non-functional requirements of Web applications and how they should be tested. From Section 7.3 onwards this chapter focuses on testing the functional requirements of Web applications. Section 7.3 presents different categories of models used to obtain suitable representations of the application to be tested. Section 7.4 presents different types of testing scopes for Web applications. In Section 7.5 several test strategies for designing test cases are discussed, while in Section 7.6 the characteristic features of tools for Web application testing are analysed. Section 7.7 shows a practical example of testing a Web application. Finally, Section 7.8 presents our conclusions and future trends.

7.2 Web Application Testing: Challenges and Perspectives

Since the Web's inception the goals and functionality offered by Web applications, as well as the technologies used to implement them, have changed considerably. Early Web applications comprised a simple set of static HTML pages. However, more recent applications offer their users a variety of functions for manipulating data, accessing databases, and carrying out a number of productive processes. These functions are usually performed by means of software components implemented by different technologies such as Java Server Pages (JSP), Java Servlets, PHP, CGI, XML, ODBC, JDBC, or proprietary technologies such as Microsoft's Active Server Pages (ASP). These components exploit a complex, heterogeneous execution environment including hardware, software, and middleware components.

The remainder of this chapter uses the term *Web application* (or simply application) to indicate the set of software components implementing the functionality and services the application provides to its users, while the term *running environment* will indicate the whole infrastructure (composed of hardware, software and middleware components) needed to execute a Web application.

The main goal of testing a Web application is to run the application using combinations of input and state to discover failures. A failure is the manifested inability of a system or component to perform a required function within specified performance requirements [13]. Failures can be attributed to faults in the application's implementation. Generally, there will be failures due mainly to faults in the application itself and failures that will be mainly caused by faults in the running environment or in the interface between the application and the environment on which it runs. Since a Web application is strictly interwoven to its running environment, it is not possible to test it separately to find out exactly what component is responsible for each exhibited failure. Therefore, different types of testing have to be executed to uncover these diverse types of failures [17].

The running environment mainly affects the non-functional requirements of a Web application (e.g. performance, stability, compatibility), while the application is responsible for the functional requirements. Thus, Web application testing has to be considered from two distinct perspectives. One perspective identifies the different types of testing that need to be executed to verify the conformance of a Web application with specified *non-functional* requirements. The other perspective considers the problem of testing the *functional* requirements of an application. It is necessary that an application be tested from both perspectives, since they are complementary and not mutually exclusive.

Questions and challenges that characterise both testing perspectives will be analysed in the next sub-sections.

7.2.1 Testing the Non-functional Requirements of a Web Application

There are different non-functional requirements that a Web application, either explicitly or implicitly, is usually required to satisfy. For each non-functional requirement, testing activities with specific aims will have to be designed. A description of the verification activities that can be executed to test the main non-functional requirements of a Web application are presented below.

Performance Testing

Performance testing is carried out to verify specified system performance (e.g. response time, service availability). Usually, performance testing is executed by simulating hundreds, or even more, simultaneous user accesses over a defined time interval. Information about accesses is recorded and then analysed to estimate the load levels exhausting the system resources.

In the case of Web applications, system performance is a critical issue because Web users do not want to wait too long for a response to their requests; as well, they also expect that services will always be available.

Effective performance testing of Web applications is a critical task because it is not possible to know beforehand how many users will actually be connected to a real-world running application. Thus, performance testing should be considered as an everlasting activity to be carried out by analysing data from access log files, in order to tune the system adequately.

Failures that can be uncovered by performance testing are mainly due to running environment faults (e.g. scarce resources, poorly deployed resources), even if any software component of the application level may contribute to inefficiency, i.e. components implementing any business rule by algorithms that are not optimised.

Load Testing

Load testing is often used as a synonym for performance testing but it differs from the latter because it requires that system performance be evaluated with a predefined load level. It aims to measure the time needed to perform several tasks and functions under predefined conditions. These predefined conditions include the minimum configuration and the maximum activity levels of the running application. Also, in this case, numerous simultaneous user accesses are simulated. Information is recorded and, when the tasks are not executed within predefined time limits, failure reports are generated.

As for the difficulties of executing load testing of Web applications, considerations similar to the ones made for performance testing can also be taken into account. Failures found by load testing are mainly due to faults in the running environment.

Stress Testing

Stress testing is conducted to evaluate a system or component at or beyond the limits of its specified requirements. It is used to evaluate the system's response at activity peaks that can exceed system limitations, and to verify if the system crashes or is able to recover from such conditions. Stress testing differs from performance and load testing because the system is executed on or beyond its breaking point, while performance and load testing simulate regular user activity.

In the case of Web applications, stress testing difficulties are similar to those that can be met in performance and load testing. Failures found by stress testing are mainly due to faults in the running environment.

Compatibility Testing

Compatibility testing is carried out to determine if an application runs as expected on a running environment that has various combinations of hardware, software, and middleware.

In the case of Web applications, compatibility testing will have to uncover failures due to the usage of different Web server platforms or client browsers, and corresponding releases or configurations.

The large variety of possible combinations of all the components involved in the execution of a Web application does not make it feasible to test them all; thus usually only the most common combinations are considered. As a consequence, just a subset of possible compatibility failures might be uncovered.

Both the application and the running environment can be responsible for compatibility failures. A general rule for avoiding compatibility failures is to provide Web application users with appropriate information about the expected configuration of the running environment and with appropriate diagnostic messages to deal with any incompatibilities found.

Usability Testing

Usability testing aims to verify to what extent an application is easy to use. Usually, design and implementation of the user interface both affect usability. Thus, usability testing is mainly centred around testing the user interface: issues concerning the correct content rendering (e.g. graphics, text editing format) as well as the clarity of messages, prompts, and commands that are to be considered and verified.

Usability is a critical issue for a Web application. Indeed, it may determine the success of the application. As a consequence, an application's front-end and the way users interact with it often are aspects that are given greater care and attention during the application's development process.

When Web application usability testing is carried out, issues related to an application's navigation completeness, correctness, and conciseness are also considered and verified. This type of testing should be an everlasting activity carried out to improve the usability of a Web application; techniques of user profiling are usually used to reach this aim.

The application is mainly responsible for usability failures.

Accessibility Testing

Accessibility testing can be considered a particular type of usability testing whose aim is to verify that the access to an application's content is allowed even in the presence of reduced hardware and software configurations on the client side (e.g. browser configurations disabling graphical visualisation, or scripting execution), or in the presence of users with disabilities, such as visual impairment.

In the case of Web applications, accessibility rules such as the one provided by the Web Content Accessibility Guidelines [24] have been established, so that accessibility testing represents verification the compliance of an application with such rules. The application itself is generally the main cause of accessibility problems, even when accessibility failures may be due to the configuration of the running environment (e.g. browsers where the execution of scripts is disabled).

Security Testing

Security testing aims to verify the effectiveness of the overall Web application's defences against undesired access of unauthorised users, its capability to preserve system resources from improper use, and granting authorised users access to authorised services and resources. Application defences have to provide protection mechanisms able to avoid or reduce damage due to intrusions, with costs that should be significantly less than damages caused by a security break.

Application vulnerabilities affecting security may be contained in the application code, or in any of the different hardware, software, and middleware components. Both the running environment and the application can be responsible for security failures.

In the case of Web applications, heterogeneous implementations and execution technologies, together with the very large number of possible users and the possibility of accessing them from anywhere, can make Web applications more vulnerable than traditional applications and security testing more difficult to accomplish.

7.2.2 Testing the Functional Requirements of a Web Application

Testing the functional requirements of an application aims at verifying that an application's features and operational behaviour correspond to their specifications. In other words, this type of testing is responsible for uncovering application failures due to faults in the functional requirements' implementation, rather than failures due to the application's running environment. To achieve this aim, any failures due to the running environment should be avoided, or reduced to a minimum. Preliminary assumptions about the running environment will have to be made before test design and execution.

Most methods and approaches used to test the functional requirements of "traditional" software can also be used for Web applications. Similarly to traditional software testing, a Web application's functionality testing has to rely on the following basic aspects:

- *Testing levels*, which specify the different scope of the tests to be carried out, i.e. the collections of components to be tested.
- *Test strategies*, which define heuristics or algorithms to create test cases from software representation models, implementation models, or test models.
- *Test models*, which represent the relationships between a representation's elements or a component's implementation [3].
- *Testing processes*, which define the flow of testing activities, and other decisions such as when to start testing, who is to perform the testing, how much effort should be used, etc.

However, despite their similarity to conventional applications, Web applications also have distinguishing features that cause specific problems for each aspect described in the previous list. For example, the definition of *testing levels* for a Web application requires greater attention than that applied to traditional software. At the *unit testing* level, the scope of a unit test cannot be defined uniquely, since it depends on the existence of different types of components (e.g. Web pages, script functions, embedded objects) residing on both the client and server side of an application. In relation to *integration testing*, the numerous different mechanisms used to integrate an application's heterogeneous and distributed components can generate several coupling levels and data flow between the components, which have to be considered to establish a correct integration strategy.

As for the *strategies for test design*, the classical approaches of black box, white box, or grey box testing may be taken into account for designing test cases, provided that preliminary considerations are defined. In general, Web applications' *black box testing* will not be different from software applications' black box testing. In both cases, using a predetermined coverage criterion, an adequate set of test cases is defined based upon the specified functionality of the item to be tested. However, a Web application's specific features can affect test design and execution. For example, testing of components dynamically generated by the running application can be very expensive, due to the difficulty of identifying and regenerating the same conditions that produced each component. Therefore, traditional testing models used to represent the behaviour of an application may have to be adapted to these characteristics and to the Web applications' running environment.

White box testing, irrespective of an application's nature, is usually based on coverage criteria that take into account structural features of the application or its components. Adequate models representing an application or component's structure are used, and coverage criteria and test cases are appropriately specified. The aim of white box testing is to cover the structural elements considered. Since the architecture and components of a Web application are largely different from those of a traditional application, appropriate models representing structural information at different levels of granularity and abstraction are needed, and coverage criteria have to be defined accordingly. For example, models representing navigation as well as traditional structural aspects of an application need to be taken into account. Coverage criteria must focus both on hyperlinks, which allow user navigation in the application, and on inner items of an application's component (e.g. its code statements).

Besides black and white box testing, *grey box* testing can also be considered for Web applications. Grey box testing is a mixture of black and white box testing, and considers both the application's behaviour, from the end user's viewpoint (same as black box testing), and the application's inner structure and technology (same as white box testing). According to [17], grey box testing is suitable for testing Web applications because it factors in high-level design, environment, and interoperability conditions. It is expected that this type of testing will reveal problems that are not easily identified by black box or white box analysis, in particular problems related to end-to-end information flow and distributed hardware/software system configuration and compatibility. Context-specific failures relevant to Web applications are commonly uncovered using grey-box testing.

Finally, for the *testing processes*, the classical approach for testing execution that starts from unit test and proceeds with integration, system testing, and acceptance testing can also be taken into account for Web applications. For each phase, however, differences with respect to testing traditional software have to be detected and specific solutions have to be designed. An important testing process issue is, for instance, to set up an *environment* to execute tests at each phase: driver or stub modules are usually required to run tests at the unit or integration phase. Solutions for testing a Web application have to explicitly consider the application's distributed running environment, and to adopt the necessary communication mechanisms for executing the components being tested.

7.3 Web Application Representation Models

In software testing the need for models that represent essential concepts and relationships between items being tested has been documented [3]. Models are able to support the selection of effective test cases, since they can be used to express required behaviour or to focus on aspects of an application's structure believed to have defects.

With regard to Web applications, models for representing their behaviour or structure have been provided by several Web application development methodologies, which have extended traditional software models to explicitly represent Web-related software characteristics. Examples of such models include the *Relationship Management Data Model (RMDM)* used by the Relationship Management Methodology (RMM) [14], which uses entity–relationship-based diagrams to describe objects and navigation mechanisms of Web applications. Other methodologies, such as Object Oriented Hypermedia (OOH) [9], integrate the traditional object-oriented models with a *navigational view* and a *presentation view* of the application. The Object-Oriented Hypermedia Design Model (OOHDM) methodology [22] allows for the construction of customised Web applications by adopting object-oriented primitives to build the application's conceptual, navigational, and interface models. WebML (Web Modelling Language) [2] is, moreover, a specification language that proposes four types of models, *Structural Model*, *Hypertext Model*, *Presentation Model*, and *Personalisation Model*, used to specify different characteristics of complex Web applications, irrespective of their implementation details. Finally, an extension of UML diagrams with new class stereotypes for representing specific Web application components, such as HTML pages, forms, server pages, is proposed in [4].

In addition to these models, other representation models explicitly geared towards Web application testing have been proposed in the literature. Two categories are currently used to classify these models: *behaviour models* and *structural models*. The former are used to describe the functionality of a Web application irrespective of its implementation. The latter are derived from the implementation of the application.

Behaviour models support black box (or responsibility-based) testing. Use case models and decision tables [6], and state machines [1], have been used to design Web application test cases for black-box testing techniques.

Structural models are used for white box testing. Both control flow representation models of a Web application's components [16,18,19], and models describing an application's organisation in terms of Web pages and hyperlinks, have been proposed [6,19]. Further details of these representations are given in Sect. 7.5.

The meta-model of a Web application [7] is now described. This model is presented in Fig. 7.1 using a UML class diagram where various types of classes and associations represent several categories of a Web application's components and their relationships. A Web application can be modelled using a UML class diagram model instantiated from this meta-model.

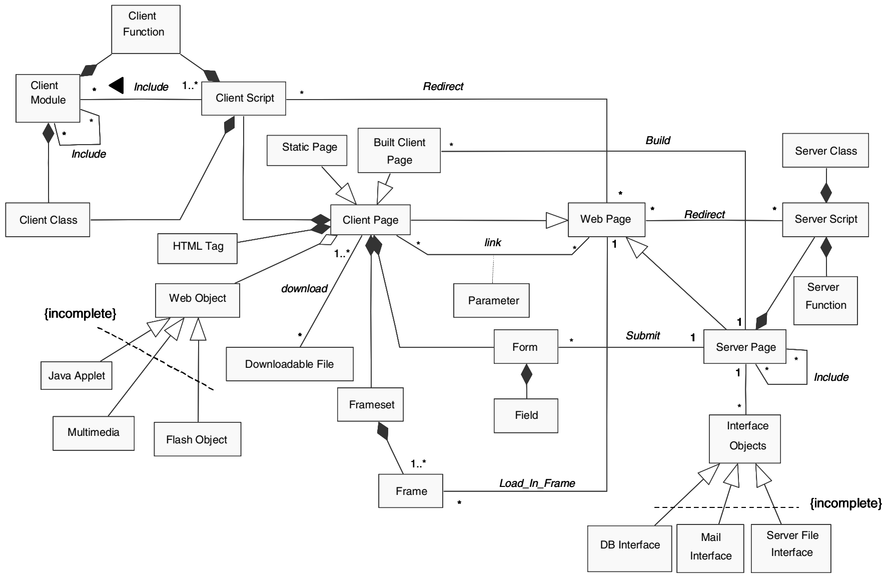

Fig. 7.1. The meta-model of a Web application presented in [7]

The meta-model assumes that a Web application comprises *Web Pages*, which can be grouped as *Server Pages*, i.e. pages that are deployed on the Web server, and *Client Pages*, i.e. pages that a Web server sends back in answer to a client request. As for the *Client Pages*, they can be classified as *Static Pages*, if their content is fixed and stored permanently, or *Client Built Pages*, if their content varies over time and is generated on-the-fly by a *Server Page*. A *Client Page* is composed of *HTML Tags*. A *Client Page* may include a *Frameset*, composed of one or more *Frames*, and in each *Frame* a different *Web Page* can be loaded. *Client Pages* may include finer-grained items implementing processing actions, such as *Client Scripts*. A *Client Page* may also include other *Web Objects* such as *Java Applets*, images and *Multimedia* Objects (e.g. sounds, movies), *Flash Objects* etc. A *Client Script* may include *Client Modules*. Both *Client Scripts* and *Client Modules* can include *Client Functions*, or C*lient Classes*. A *Client Script* may redirect the elaboration to another *Web Page*. In addition, a *Client Page* may be linked to another *Web Page*, through a hyperlink to the *Web Page*'s URL: a link between a *Client Page* and a *Web Page* may be characterised by any *Parameter* that the *Client Page* provides to the *Web Page*. A *Client Page* may also be associated with any *Downloadable File*, or it may include any *Form*, composed of different types of *Field* (e.g. select, button, text-area fields). *Forms* are used to collect user input and to *submit* the input to the *Server Page* responsible for its elaboration. A *Server Page* may be composed of any *Server Script*, which can include any S*erver Class* or *Server Function*, implementing any processing action, which may either *redirect* the request to another *Web Page*, or dynamically *build* a *Client Built Page* providing the result of an elaboration. Finally, a *Server Page* may include other *Server Pages*, and may be associated with other *Interface Objects* allowing the connection of the *Web application* to a DBMS, a file server, a mail server, or another system.

7.4 Unit Integration and System Testing of a Web Application

The flow of activities of a software testing process usually begins with unit testing and proceeds with integration and system test. The aim of unit testing is to verify each application's individual source code component, while integration testing considers combined parts of an application to verify how they function together. Finally, system testing aims at discovering defects that are properties of the entire system rather than of its individual components.

7.4.1 Unit Testing

To set up Web application unit testing it is important to choose the application components to be tested individually. If we consider the model of a Web application as presented in Fig. 7.1, different types of unit may be identified (e.g. Web pages, scripting modules, forms, applets, servlets). However, the basic unit that can actually be tested is a Web page, if we consider that any page's element should also automatically be considered for testing. As a consequence, *pages* are usually considered at the unit testing level, although there are some differences between testing a client or a server page. We present these differences below.

Testing Client Pages

Client pages constitute the application's user interface. They are responsible for showing textual information and/or hyperlinks to users, for accepting user input, and for allowing user navigation throughout the application. A client page may include scripting code modules that perform simple functions, such as input validation or simple computations. Moreover, client pages may be decomposed into several frames in which other client pages can be visualised.

Testing a client page (including just HTML code) aims to verify:

- Compliance of the content displayed by the page to the one specified and expected by a user (e.g. the rendering in the browser of both textual content and its formatting of forms, images and other Web objects will have to be verified).
- Correctness of target pages pointed to by hyperlinks, i.e. when a link is selected, the right page should be returned.
- Existence of pending links, i.e. links to pages that do not exist.
- Correctness of the actions performed when a button, or any other active object, is selected by a user.
- Correctness of the content visualised in the frames.

If the client page includes scripting code, failures due to scripts will also have to be verified.

Testing dynamically generated client pages (built-in pages) is a particular case of client page testing. The basic problem with this testing is that the availability of built-in pages depends on the ability to identify and repeat the same conditions (in terms of application state and user input) used to build such pages. A second problem is that of having too many pages being generated, since the number of dynamic pages can be considerable, depending on the large number of possible combinations of application state and user input. Equivalence class partitioning criteria (such as those considering exemplar path execution of server pages) should be used to deal with this issue.

Unit testing of client pages can be carried out using white box, black box, or grey box testing techniques. Several implementation-based criteria can be used to evaluate white box test coverage, such as:

- HTML statement coverage.
- Web object coverage, i.e. each image, multimedia component, applet, etc. will have to be tested at least once.
- Script block coverage, i.e. each block of scripting code, such as client side functions, will have to be executed at least once.
- Statement/branch/path coverage for each script module.
- Link coverage.

Testing Server Pages

The main goal of server pages is to implement an application's business logic, thus coordinating the execution of business rules and managing the storing and retrieving of data into/from a database.

Usually, server pages are implemented by a mixture of technologies, such as HTML, script languages (e.g. VBS, JSP), Java servlets, or COTS.

Typical results of server page execution are data storage into a database, or generation of client pages based on user requests.

Testing a server page aims to identify failures of different types, such as:

- Failures in the executions of servlets, or COTS.
- Incorrect executions of data being stored into a database.
- Failures due to the existence of incorrect links between pages.
- Defects in dynamically generated client pages (such as non-compliance of the client page with the output specified for the server page).

Unit testing of server pages can also be carried out using white box, black box, or grey box techniques. White box coverage criteria include:

- Statement/branch/path coverage in script modules.
- HTML statement coverage.
- Servlet, COTS, and other Web object coverage.
- Hyperlink coverage.
- Coverage of dynamically generated pages.

Appropriate driver and stub pages have to be generated to carry out unit page testing effectively (see Sect. 7.6 for a discussion on the generation of such drivers and stubs).

7.4.2 Integration Testing

Integration testing is the testing of a Web application's combined pages to assess how they function together. An integration criterion has to be used to choose the pages to be combined and tested together. Design documentation showing relationships between pages can be used to define an integration strategy.

As an example, the Web application model, obtained by instantiating the meta-model presented in Fig. 7.1, can be used to identify the pages to be combined. Pages chosen will be those linked by direct relationships, such as *hyperlinks*, or by dependency relationships due to *redirect* or *submit* statements (included either in a server or in a client page), or by *build* relationships between a server page and the client page produced.

Another integration criterion may consider a server page and each client page it generates at run time as a unit to be tested. The problem of client page explosion will have to be addressed with equivalence class partitioning criteria.

Page integration can be driven by the use cases implemented by the application, or any other description of the application's functional requirements. For each use case (or functional requirement), Web pages collaborating for its implementation are to be considered for integration purposes. The identification of such Web pages can be made by analysing development documentation or by reverse engineering the application code. Reverse engineering techniques, such as the one described in [5], can be used to analyse the relationships between pages and to identify clusters of interconnected pages that implement a use case.

At the integration testing level, both the behaviour and the structure of the Web application will have to be considered: knowledge of the application structure will be used to define the set of pages to be integrated, while knowledge of the behaviour implemented by these pages will be needed to carry out integration testing with a black box strategy. Therefore, grey box techniques may be more suitable than pure black or white box ones to carry out integration testing.

7.4.3 System Testing

System testing aims to discover defects related to the entire Web application. In traditional software testing, black box approaches are usually exploited to accomplish system testing and to identify failures in the externally visible behaviour of the application. However, grey box techniques that consider the application navigation structure, in addition to its behaviour, for designing test cases may be more effective in revealing Web application failures due to incorrect navigation links among pages (such as links connecting a page to a different one from the specified page, pending links, or links to unreachable pages).

Depending on the testing strategy adopted, coverage criteria for system testing will include:

– User functions/use cases coverage (if a black box approach is used).
– Page (both client and server) coverage (usable for white box or grey box approaches).
– Link coverage (usable for white box or grey box approaches).

7.5 Strategies for Web Application Testing

Testing strategies define the approaches for designing test cases. They can be responsibility based (also known as black box), implementation based (or white box), or hybrid (also known as grey box) [3]. Black box techniques design test cases on the basis of the specified functionality of the item to be tested. White box techniques rely on source code analysis to develop test cases. Grey box testing designs test cases using both responsibility-based and implementation-based approaches.

This section discusses representative contributions presented in the literature for white box, black box, and grey box testing of Web applications.

7.5.1 White Box Strategies

White box strategies design test cases on the basis of a code representation of the component under test (i.e. the *test model*), and of a *coverage model* that specifies the parts of the representation that must be exercised by a test suite. As an example, in the case of traditional software the control flow graph is a typical test model, while statement coverage, branch coverage, or basis-path coverage are possible code coverage models.

As for the code representation models adopted to test Web applications, two main families of structural models are used: the first one focuses on the level of abstraction of single statements of code components of the application, and represents the traditional information about their control flow or data flow. The second family considers the coarser degree of granularity of the pages of the Web application and essentially represents the navigation structure between pages of the application with eventual additional details. As the coverage criteria, traditional ones (such as those involving nodes, edges, or notable paths from the graphical representations of these models) have been applied to both families of models.

Two white box techniques proposed in the literature to test Web applications will be presented in this section. The first technique was proposed by Liu et al. [17] and exploits a test model that belongs to the first family of models, while the second one was proposed by Ricca and Tonella [19, 20] and is based on two different test models, each one belonging to a different family.

The white box technique proposed by Liu et al. [17] is an example of how data-flow testing of Web applications can be carried out. The approach is applicable to Web applications implemented in the HTML and XML languages, including interpreted scripts as well as other kinds of executable components (e.g. Java applets, ActiveX controls, Java beans) at both the client and server side of the application.

The approach is based on a Web application test model, WATM, that includes an object model, and a structure model. The o*bject model* represents the heterogeneous components of a Web application and the ways they are interconnected using an object-based approach. The model includes three types of objects (i.e. client pages, server pages, and components) and seven types of relationships between objects. Each object is associated with attributes corresponding to program variables or other HTML specific document elements (e.g. anchors, headers, or input buttons), and operations corresponding to functions written in scripting or programming languages. Relationships between objects are of seven types: inheritance, aggregation, association, request, response, navigation, and redirect. The first three have the classical object-oriented semantics, while the last four represent specific relationships between client and server pages. A *request* relationship exists between a client and a server page when a server page is requested by a client page; a *response* relationship exists between a client and a server page when a client page is generated by a server page as a response of an elaboration; for two client pages there is a *navigation* relationship if one of them includes a hyperlink to the other page; finally, between two server pages there is a *redirect* relationship if one of them redirects an HTTP request to the other. The s*tructure model* uses four types of graphs to capture various types of data flow information on a Web application: the *Control Flow Graph* (CFG) of an individual function, the *Interprocedural Control Flow Graph* (ICFG) that involves more than one function and integrates the CFGs of functions that call each other, the *Object Control Flow Graph* (OCFG) that integrates the CFGs of object functions that are involved in sequences of function invocations triggered by GUI events, and, finally, the *Composite Control Flow Graph* (CCFG) that captures the pages where a page passes data to the other one when the user clicks a hyperlink, or submits a form, and is constructed by connecting the CFGs of the interacting Web pages.

The data flow testing approach derives test cases from three different perspectives: intra-object, inter-object, and inter-client. For each perspective, def-use chains of variables are taken into account for defining test paths that exercise the considered def-use chains. Five testing levels specifying different scopes of the tests to be run have been defined, namely: Function, Function Cluster, Object, Object Cluster, and Application level.

For the *intra-object* perspective, test paths are selected for variables that have def-use chains within an object. The def-use chains are computed using the control flow graphs of functions included in the object, and can be defined at three different testing levels: single *function*, *cluster of functions* (i.e. set of functions that interact via function calls within an object), and *object level* (considering different sequences of function invocations within an object).

For the *inter-object* perspective, test paths are selected for variables that have def-use chains across objects. Def-use chains have to be defined at the *object cluster* level, where each cluster is composed by a set of message-passing objects.

Finally, the *inter-client* perspective derives test paths on the basis of def-use chains of variables that span multiple clients, since in a Web application a variable can be shared by multiple clients. This level of testing is called *application* level.

This testing technique is relevant since it represents a first attempt to extend the data flow testing approaches applicable to traditional software to the field of Web applications. However, to make it actually usable in real-world Web application testing, further investigation is required. Indeed, the effectiveness of the technique has not been validated by any experiment involving more than one example Web application: to carry out these experiments, an automated environment for testing execution, including code analysers, data flow analysers, and code instrumentation tools, would be necessary. Moreover, indications about how this data flow testing approach may be integrated in a testing process would also be needed: as an example, the various testing perspectives and levels proposed by the approach might be considered in different phases of a testing process to carry out unit test, as well as integration or system test. However, in this case an experimental validation and tuning would also be required.

A second proposal in the field of structural testing of Web applications has been suggested by Ricca and Tonella [19], who proposed a first approach for white box testing of primarily static Web applications. This approach was based on a test model named the n*avigational model* that focuses on HTML pages and navigational links of the application. Later, the same authors presented an additional lower layer model, the c*ontrol flow model*, representing the internal structure of Web pages in terms of the execution flow followed [20]. This latter model has also been used to carry out structural testing.

In the navigational model two types of HTML pages are represented: static pages, whose content is immutable, and dynamic pages, whose content is established at run time by server computation, on the basis of user input and server status. Server programs (such as scripts or other executable objects) running on the server side of the application, and other page components that are relevant for navigational purposes, such as forms and frames, are also part of the model. Hyperlinks between HTML pages and various types of link between pages and other model components are included in this code representation.

As for the control flow model, it takes into account the heterogeneous nature of statements written in different coding languages, and the different mechanisms used to transfer control between statements in a Web application. It is represented by a directed graph whose nodes correspond to statements that are executed either by the Web server or by the Internet browser on the client side, and whose edges represent control transfer. Different types of nodes are shown in this model, according to the programming language of the respective statements.

A *test case* for a Web application is defined as a sequence of pages to be visited, plus the input values to be provided to pages containing forms. Various coverage criteria applicable to both models have been proposed to design test cases: they include p*ath coverage* (requiring that all paths in the Web application model are traversed in some test case), b*ranch coverage* (requiring that all branches in the model are traversed in some test case), and n*ode coverage* (requiring that all nodes in the model are traversed in some test case).

Assuming that the nodes of the representation models can be annotated by definitions or uses of data variables, further data flow coverage criteria have been described too: a*ll def-use* (all definition-clear paths from every definition to every use of all Web application variables are traversed in some test case), a*ll uses* (at least one def-clear path if any exists from every definition to every use of all Web application variables traversed in some test case), a*ll defs* (at least one def-clear path if any exists from every definition to at least one use of all Web application variables is traversed in some test case).

This testing approach is partially supported by a tool, ReWeb, that analyses the pages of the Web application and builds the corresponding navigational model, and another tool, TestWeb, that generates and executes test cases. However, the latter tool is not completely automated, since user intervention is required to generate input and act as an oracle. The main limitation of this testing approach concerns its scalability (consider the problem of path explosion in the presence of cycles on the graphs, or the unfeasibility of the all-do coverage criterion).

A few considerations about the testing levels supported by white box techniques can be made. Some approaches are applicable at the unit level, while others are considered at the integration and system levels. For instance, the first approach proposed by Liu et al. [17] is applicable at various testing levels, ranging from unit level to integration level. As an example, the intra-object perspective can be used to obtain various types of units to be tested, while inter-object and inter-application perspectives can be considered for establishing the items to be tested at the integration level. Conversely, the approaches of Ricca and Tonella are applicable exclusively at the system level. As a consequence, the choice of a testing technique to be applied in a testing process will also depend on the scope of the test to be run.

7.5.2 Black Box Strategies

Black box techniques do not require knowledge of software implementation items under test since test cases are designed on the basis of an item's specified or expected functionality.

One main issue with black box testing of Web applications is the choice of a suitable model for specifying the behaviour of the application to be tested and to derive test cases. Indeed, this behaviour may significantly depend on the state of data managed by the application and on user input, with the consequence of a state explosion problem even in the presence of applications implementing a few simple requirements.

Solutions to this problem have been investigated and presented in the literature. Two examples of proposed solutions are discussed in this subsection. The first example is offered by the black box testing approach proposed by Di Lucca et al. [6] that exploits decision tables as a combinatorial model for representing the behaviour of a Web application and to produce test cases. The second example is provided by Andrews et al. [1] where state machines are proposed to model state-dependent behaviour of Web applications and to design test cases.

Di Lucca et al. [6] suggest a two-stage black box testing approach. The first stage addresses *unit testing* of a Web application, while the second stage considers *integration testing*. The scope of a unit test is a single application page, either a client or server page, while the scope of an integration test is a set of Web pages that collaborate to implement an application's use case.

Unit test is carried out with a responsibility-based approach that uses *decision tables* to represent page requirements, and therefore derive test cases. A decision table can be used to represent the behaviour of software components whose responses are each associated with a specific condition. Usually a decision table has two parts: the *condition section* (listing conditions and combinations of conditions) and the *action section* (listing responses to be produced when corresponding combinations of conditions are true). Each unique combination of conditions and actions is a *variant*, represented as a single row in the table.

As for the *unit testing* of client and server pages, the approach requires that each page under test is preliminarily associated with a decision table describing a set of variants of the page. Each variant represents an alternative behaviour offered by the page and is defined in terms of an Input section and an output section. In the case of client pages, the *input section* describes a condition in terms of i*nput variables* to the page, *input actions*, and *state before test* where the state is defined by the values assumed, before test execution, by page variables, tag attributes, cookies, and by the state of other Web objects used by page scripts. In the o*utput section,* the action associated with each condition is described by the e*xpected results*, e*xpected output* actions, and e*xpected state after test* (defined as for the state before test). Table 7.1 shows the template of the decision table for client page testing.

Such specification technique may be affected by the problem of variant explosion. However, criteria for partitioning input section data into equivalence classes may be defined and used to reduce the set of variants to be taken into account.

In the case of server pages, the decision table template is slightly different (see Table 7.2): for each page variant the i*nput section* includes the *input variables* field that comprises the variables provided to the server page when it is executed, and the s*tate before test* field that is defined by the values assumed, before test execution, by page session variables and cookies, as well as by the state of the session objects used by the page scripts. In the o*utput section,* the e*xpected results* field represents the values of the output variables computed by the server page scripts, the e*xpected output* field includes the actions performed by the server side scripts (such as composing and sending an e-mail message), and the e*xpected state after test* field includes the values of variables and cookies, as well as the state of session objects, after execution.

Table 7.1. A decision table template for client page testing

Variant	Input Section			Output Section		
	Input variables	Input actions	State before test	Expected results	Expected output actions	Expected state after test
…	…			…		

Table 7.2. A Decision Table template for server page testing

Variant	Input Section		Output Section		
	Input variables	State before test	Expected results	Expected output actions	Expected state after test
…			…		

As for the definition of the decision tables, the authors propose to compile them by analysing the development documentation (if available) or by reverse engineering the Web application code, and focusing on the page inner components that help to define the conditions and actions of each variant. An object model of a Web application representing each component of the application relevant for testing purposes is specifically presented by the authors to support this type of analysis. This model is actually an extended version of the one reported in Fig. 7.1, including additional relevant details for the aims of testing (such as session variables).

The *test case selection strategy* is based on the decision tables and requires that test cases are defined in order to cover each table variant for both true and false values. Other criteria based on partitioning the input sets into equivalence classes are also suggested for defining test cases.

In this testing approach, decision tables are also used to develop *driver* and *stub* modules which will be needed to execute the client page testing. A driver module will be a Web page that interacts with the client page by populating its input forms and generating the events specified for the test case. The driver page will include script functions, and the Document Object Model (DOM) will allow its interaction with the tested page. Stub modules can be developed as client pages, server pages or Web objects. The complexity of the stub will depend both on the type of interaction between the tested page and the component to be substituted, and on the complexity of the function globally implemented by the pair of components.

As for the *integration testing*, a fundamental question is the one of determining which Web pages have to be integrated and tested. The authors of this approach propose to integrate Web pages that collaborate with the implementation of each use case (or functional requirement) of the application. They propose to analyse the object model of the Web application in order to find client and server pages to be gathered together. A valuable support for the identification of clusters of interconnected pages may be provided by *clustering techniques*, such as the one proposed in [5]. This technique produces clusters of pages on the basis of a measure of coupling of interconnected pages that associates different weights to different types of relationship (*Link*, *Submit*, *Redirect*, *Build*, *Load_in_Frame*, *Include*) between pages. Once clusters have been defined and use cases have been associated to each of them, the set of pages included in each cluster will make up the item to be tested. For each use case a decision table can be defined to drive integration testing. Such a decision table can be derived from the ones defined for the unit testing of the single pages included in the cluster.

The second black box approach for Web application testing considered in this section exploits *Finite State Machines* (FSMs) for modelling software behaviour and deriving test cases from them [1]. This approach explicitly takes into account the state-dependent behaviour of Web applications, and proposes specific solutions for addressing the problem of state explosion.

The process for test generation comprises two phases: in the first phase, the Web application is modelled by a hierarchical collection of FSMs, where the bottom-level FSMs are formed by Web pages and parts of Web pages, while a top-level FSM represents the whole application. In the second phase, test cases are generated from this representation.

The model of the Web application is obtained as follows. First, the application is partitioned into clusters that are collections of Web pages and software modules that implement a logical function. This clustering task is made manually and is thus subjective. Second, Web pages that include more than one HTML form, each of which is connected to a different back-end software module, will be modelled as multiple Logical Web Pages (LWP), in order to facilitate testing of these modules. Third, an FSM will be derived for each cluster, starting from bottom-level clusters containing only modules and Web pages (no clusters), and therefore aggregating lower-level FSMs into a higher level FSM. Ultimately, an Application FSM (AFSM) will define an FSM of the entire Web application. In each FSM, nodes will represent clusters and edges will represent valid navigation among clusters. Moreover, edges of the FSMs will be annotated with inputs and constraints that may be associated with the transitions. Constraints on input, for instance, will indicate if input data are optional and their eventual input order. Information will also be propagated between lower-level FSMs.

Annotated FSMs and aggregate FSMs are thus used to generate tests. Tests are considered as sequences of transitions in an FSM and the associated constraints. Test sequences for lower-level FSMs are combined to form the test sequences for the aggregate FSMs. Standard graph coverage criteria, such as *all nodes* and *all edges,* are used to generate sequences of transitions for clusters and to aggregate FSMs.

While the approach of Di Lucca et al. provides a method for both unit and integration testing, the one by Andrews et al. mainly addresses integration and system testing. Both approaches use clustering to identify groups of related pages to be integrated, even if in the second one the clustering is made manually, and this may limit the applicability of the approach when large-size applications are tested.

The second method can be classified as a grey box rather a than pure black box technique. Indeed, test cases are generated to cover all the transitions among the clusters of LWPs, and therefore knowledge of the internal structure of the application is needed. Grey box testing strategies will be discussed in the next subsection.

7.5.3 Grey Box Testing Strategies

Grey box testing strategies combine black box and white box testing approaches to design test cases: they aim at testing a piece of software against its specification but using some knowledge of its internal workings.

Among the grey box strategies we will consider the ones based on the collection of user session data. These methods can be classified as grey box since they use collected data to test the behaviour of the application in a black box fashion, but they also aim at verifying the coverage of any internal component of the application, such as page or link coverage.

Two approaches based on user session data will be described here.

7.5.4 User Session Based Testing

Approaches based on data captured in user sessions transparently collect user interactions with the Web server and transform them into test cases using a given strategy.

Data to be captured about the user interaction with the Web server include clients' requests expressed in form of URLs and name value pairs. These data can be obtained from the log files stored by the Web servers, or by adding script modules on the requested server pages that capture the name value pairs of exchanged parameters. Captured data about user sessions can be transformed into a set of HTTP requests, each one providing a separate test case.

The main advantage of this approach is the possibility of generating test cases without analysing the internal structure of a Web application, thus reducing the costs of finding inputs. In addition, generating test cases using user session data is less dependent on the heterogeneous and fast-changing technologies used by Web applications, which is one of the major limitations of white box testing techniques. However, it can be argued that the effectiveness of user session techniques depends on the set of user session data collected: the wider this set, the greater the effectiveness of the approach to detect faults; but the wider the user session data set, the greater the cost of collecting, analysing and storing data. Therefore there is a trade-off between test suite size and fault detection capability.

Elbaum et al. [8] propose a user session approach to test a Web application and present the results of an empirical study where the effectiveness of white box and user session techniques was compared. In the study, user session collected data consist of sequences of HTTP requests made by users. Each sequence reports the pages (both client and server ones) the user visited together with the data he/she provided as input, in addition to the data resulting from the elaboration of requests made by the user.

The study considered two implementations of the white box testing approach proposed by Ricca and Tonella [19], and three different implementations of the user session approach. The first implementation transforms each individual user session into a test case; the second implementation combines interactions from different user sessions; and the third implementation inserts user session data into a white box testing technique. The study explored the effectiveness of the techniques in terms of the fault detection they provide, the cost-effectiveness of user-session- based techniques, and the relationship between the number of user sessions and the effectiveness of the test suites generated based on those sessions' interactions. As a general result, the effectiveness of white box and user session techniques was comparable in terms of fault detection capability, even if the techniques showed it was possible to find different types of faults. In particular, user session techniques were not able to discover faults associated with rarely entered data. The experiment also showed that the effectiveness of user session techniques improves as the number of collected user sessions increases. However, as the authors recognised, the growth of this number puts additional challenges on the cost of collecting and managing sessions, such as the problem of finding an oracle to establish the expected output of each user request. The possibility of using reduction techniques, such as the one described in [10], is suggested by the authors as a feasible approach for reducing test suite size, but its applicability needs further investigation. A second empirical study carried out by the same authors and described in [8] essentially confirmed the results of the first experiment.

Sampath et al. [21] have explored the possibility of using *concept analysis* to achieve scalability in user-session based testing of Web applications. Concept analysis is a technique for clustering objects that have common discrete attributes. It is used in [21] to reduce a set of user sessions to a minimal test suite, which still represents actual executed user behaviour. In particular, a user session is considered as a sequence of URLs requested by the user, and represents a separate use case offered by the application. Starting from an original test suite including a number of user sessions, this test suite is reduced by finding the smallest set of user sessions that covers all the URLs of the original test suite. At the same time, it represents the common URL of the different use cases represented by the original test suite. This technique enables an incremental approach that updates the test suite on-the-fly, by incrementally analysing additional user sessions. The experiments carried out showed the actual test suite reduction is achievable by the approach, while preserving the coverage obtained by the original user sessions' suite, and with a minimal loss of fault detection. The authors have developed a framework that automates the entire testing process, from gathering user sessions through the identification of a reduced test suite to the reuse of that test suite for coverage analysis and fault detection. A detailed description of this framework can be found in [21].

7.6 Tools for Web Application Testing

The effectiveness of a testing process may significantly depend on the tools used to support the process. Testing tools usually automate some tasks required by the process (e.g. test case generation, test case execution, evaluation of test case results). Moreover, testing tools may support the production of useful testing documentation and its configuration management.

A variety of tools for Web application testing has been proposed, where the majority was designed to carry out performance and load testing, security testing, or to implement link and accessibility checking and HTML validation. As for the functional testing, existing tools' main contribution is limited to managing test case suites created manually, and to matching the test case results with respect to an oracle created manually. Greater support for automatic test case generation would help enhance the practice of testing Web applications. User session testing can also be useful since it captures details of user interactions with the Web application. Test scripts that automatically repeat such interactions could also be created to assess the behaviour exhibited by the application. A list of more than 200 either commercial or freeware Web testing tools for Web applications is presented in [12].

Web application testing tools can be classified using the following six main categories:

a) Load, performance and stress test tools.
b) Web site security test tools.
c) HTML/XML validators.
d) Link checkers.
e) Usability and accessibility test tools.
f) Web functional/regression test tools.

Tools belonging to categories a), b), e) can be used to support non-functional requirement testing, while tools from categories c) and d) are more oriented to verifying the conformance of a Web application code to syntactical rules, or the navigability of its structure. This functionality is often offered by Web site management tools, used to develop Web sites and applications. Tools from category f) support functionality testing of Web applications and include, in addition to capture and replay tools, other tools supporting different testing strategies such as the one we analysed in Sect. 7.5.

Focusing on tools within category f), their main characteristics are discussed below, where the main differences from tools usable for traditional applications testing are also highlighted.

Services that are generic and aim to aid the functionality testing of a Web application should include:

- *Test model generation*: this is necessary to produce an instance of the desired/ specified test model of the subject application. This model may be either one of the models already produced in the development process, and the tool will have just to import it, or produced by reverse engineering the application code.
- *Test Case Management*: this is needed to support test case design and testing documentation management. Utilities for the automatic generation of the test cases would be desirable.

- *Driver and Stub Generation*: this is required to produce automatically the code of the Web pages implementing the driver and stub modules, needed for test case execution.
- *Code Instrumentation*: this is necessary to instrument automatically the code of the Web pages to be tested, by inserting probes that automatically collect data about test case execution.
- *Test result analysis*: this service will analyse and automatically evaluate test case results.
- *Report generation*: this service will produce adequate reports about analysis results, such as coverage reports about the components exercised during the test.

A generic possible architecture of such a tool is depicted in Fig. 7.2, comprising the following main components:

- *Interface* layer: implements a user interface providing access to the functions offered by the tool.
- *Service* layer: includes the components implementing tool services.
- *Repository* layer: includes the persistent data structures storing the Web application model, test cases and test logs, and the files of the instrumented Web pages, driver Web pages, stub Web pages, as well as the test reports.

Services offered by the tool, such as driver and stub generation, as well as code instrumentation and test model generation, are more reliant on the specific technologies used to implement the Web application, while others will be largely independent of the technologies. As an example, different types of drivers and stubs will have to be generated for testing client and server Web pages as the technology (e.g. the scripting languages used to code the Web pages) affects the way drivers and stubs are developed.

In general, the driver of a client page has the responsibility of loading the client page into a browser, where it is executed, while the driver of a server page requires the execution of the page on the Web server. Stubs of a client page have to simulate the behaviour of pages that are reachable from the page under test by hyperlinks, or whose execution on the Web server is required by the page. Stubs of a server page have to simulate the behaviour of other software components whose execution is required by the server page under test. Specific approaches have to be designed to implement drivers and stubs for Web pages created dynamically at run time.

Depending on the specific technology used to code Web pages, different code instrumentation components also have to be implemented. Code analysers, including different language parsers, have to be used to identify automatically the points where probes are to be inserted in the original page code.

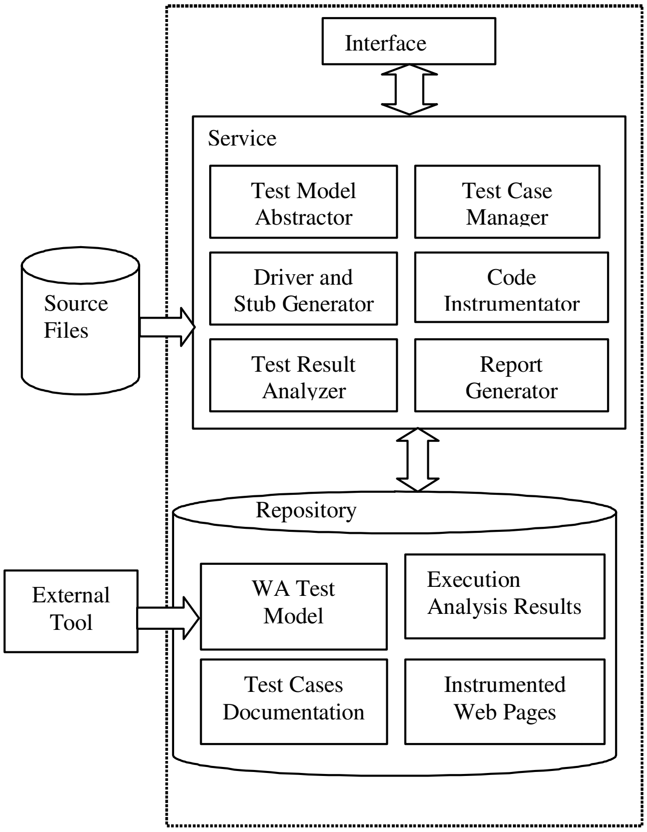

Fig. 7.2. The layered architecture of a tool supporting Web application testing

Analogously, the test model generator component that has to reverse-engineer the application code for generating the test model is largely dependent on the technologies used to implement the application. Code analysers are also required in this case.

The other modules of such a generic tool are less affected by the Web application technologies, and can be developed as in the case of traditional applications.

7.7 A Practical Example of Web Application Testing

In this section we present an example where the functional requirements of an industrial Web application are tested. It addresses the problem of testing an existing Web application using poor development documentation. We also present additional analysis techniques needed to support the application testing in the case of existing and scarcely documented Web applications. Such a problem may also exist within the development process, where the testing of scarcely documented Web applications is often encountered.

The application's unit, integration and system testing will be analysed and testing approaches proposed in the literature are used to accomplish these tasks.

The Web application presented is named "Course Management", and was developed to support the activities of undergraduate courses offered by a Computer Science Department. The application provides students and teachers with several distinct services: a teacher can publish course information, and manage examination sessions and student tutoring agendas, while students can access course information, and register for a course or an examination session. A registered student can also download teaching material.

The technologies used to implement the application are HTML, ASP, VB Script and Javascript. The application includes a Microsoft Access database, and is composed of 106 source files whose total size is close to 500 Kbytes. As for the development documentation, just a textual description of user functions was available.

The first step is to carry out a preliminary *reverse engineering* process to reconstruct design documentation that is essential for a testing activity. Such documentation includes a specification of functional requirements implemented by the application, design documentation providing the application's organisation in terms of pages and their interconnection relationships, as well as traceability information.

To obtain this information, the reverse engineering approach and the WARE tool presented in [7] were used. This tool the allows main components of a Web application and relationships between components to be automatically obtained by source code static analysis. The tool also provides the graphical representation of this information, called WAG (Web Application connection Graph). Table 7.3 lists a count of the items found in the application code for each category of components and relationships identified by the tool, while Fig. 7.3 reports the WAG depicting all the identified components and relationships. In this graph, which is an instantiation of the application's meta-model, different shapes have been used to distinguish different types of components and relationships. As an example, a box is used for drawing a Static Page, a trapezium for a Built Client Page and a diamond for a Server Page.

Using the clustering technique described in [5] and exploiting the available documentation on user functions, the application's use case model was reconstructed, and groups of pages, each implementing a use case, were identified. Figure 7.4 shows this use case model.

The testing process carried out was driven by the application's use case model. For each use case, a unit testing of the Web pages implementing the case was executed, using the black box technique based on the decision tables proposed in [6]. After the unit testing, an integration testing was carried out. In what follows, we will refer to the use case named "Teacher and course management" to show how testing was carried out.

The "Teacher and course management" use case implements the application behaviour permitting a registered teacher to manage his/her personal data and data about courses he/she teaches. This use case allows a teacher to:

– F1: register, update or delete personal data.
– F2: add a new course and associate it to the teacher for the current academic year.
– F3: update/delete the data about a course taught by the teacher.

Figure 7.5 specifies the functional requirements of the function F2. Figure 7.6 shows, using the UML extensions from Conallen [4], an excerpt of the WAG, made up by the pages implementing the function F2..

Figure 7.7 shows the rendering of the client page AddCourse.html (all the labels/prompts in the page are in Italian) including a form that allows the input of data needed to register a new course and to be added to the ones taught by the teacher in the current year.

The unit testing of this page has been carried out using the following approach. We started by analysing the responsibilities of this page. The page is in charge of visualising a form (see Fig. 7.7) that allows the input of required data, checking that all fields have been filled in, checking the validity of the academic year value, and submitting the input data to the server page AddCourse.asp. Moreover, a Reset button in the page allows to be "blanked" all the form fields while a couple of radio buttons labelled by YES/NO in the page are used to ask if the user wants to input data for more than one course.

Finally, this page automatically computes the value of the second year of the academic year field, after the first year value has been provided.

In order to associate the Web page with the decision table required by the testing approach, for each page input item (such as form fields, buttons, selection box, etc.) domain analysis was carried out to identify sets of valid and not valid values. The functional specifications reported in Fig. 7.5 were used to accomplish domain analysis, whose results are reported in Table 7.4. In the table, the input element named "More Courses?" is referred to the pair of radio buttons labelled YES/NO.

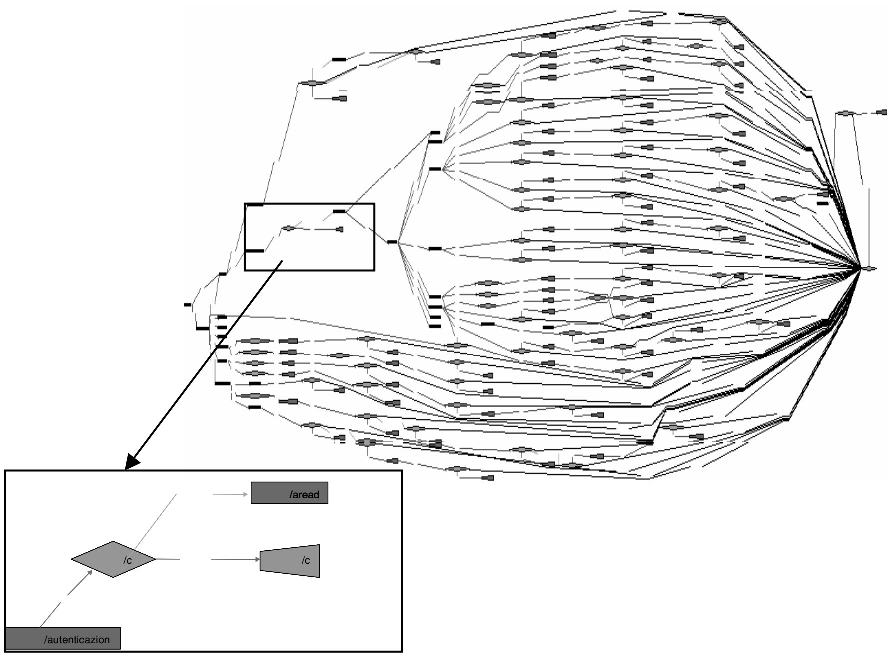

Fig. 7.3. The WAG of the “Course Management” Web application

Table 7.3. Type and count of Web application items identified by static analysis

Item type	Count
Server Page	75
Static Page	23
Built Client Page	74
Client Script	132
Client Function	48
Form	49
Server Script	562
Server Function	0
Redirect (in Server Scripts)	7
Redirect (in Client Scripts)	0
Link	45
Submit	49
Include	57
Load in Frame Operation	4

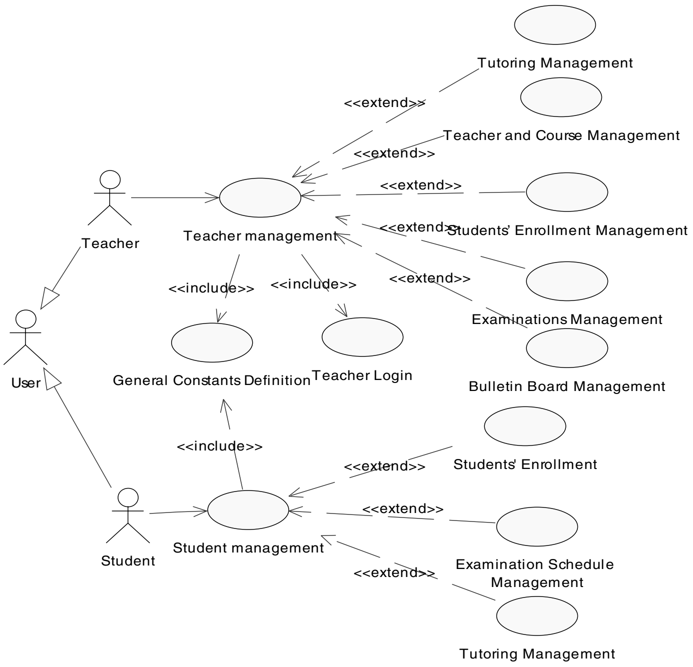

Fig. 7.4. The use case model of the "Course Management" Web application

Function: *F2* Creates a new course and associates it with a registered teacher for the current academic year.
Pre-condition: The teacher has to be already registered at the Web application.

- The teacher inputs Course Code and Name, and the Academic Year. If any datum is missing, an error message is displayed.
- The Course Code must not yet exist in the database; the Course Name may already exist in the database but associated to a different code. If the Course Code already exists, an error message is displayed.
- The Academic Year and the current Academic Year must coincide, otherwise a message error is displayed.
- If all the data are valid, the new course is added into the database and a message is sent to client to notify the success of the operation.

Post-condition: The teacher is associated with the new course for the current academic year.

Fig. 7.5. Functional requirements of function F2

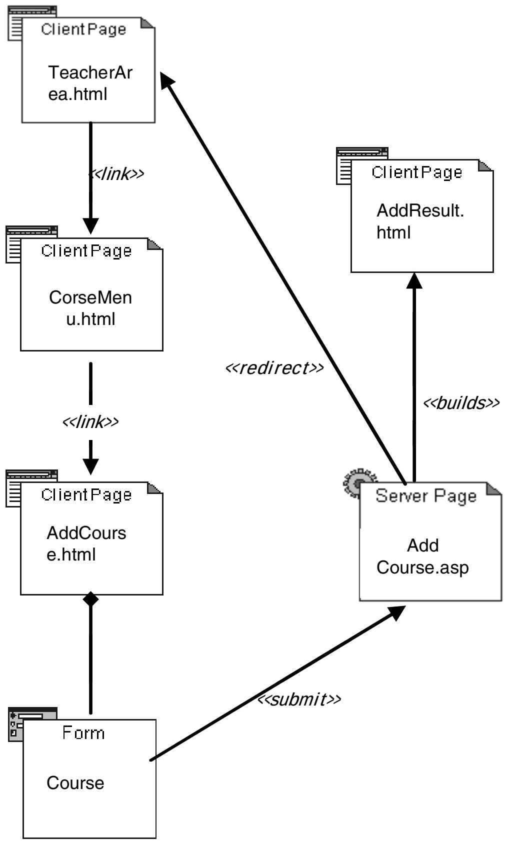

Fig. 7.6. An excerpt of the WAG using Conallen's graphical notation

Based on the specifications in Fig. 7.5 and information in Table 7.4, the decision table reported in Table 7.5 was obtained. This decision table reports all the admissible combinations, deduced by the page implementation, of valid/not valid sets of values of the input elements, together with the expected results and actions.

In Table 7.5, the columns reporting status before and after test have not been shown for the sake of readability. The status before and after the test was specified with respect to the status of the database. For each variant, the state before test is always: "The teacher is registered in the database and is allowed to do this operation". The status after test is "The execution of the test cases does not change the status of the database".

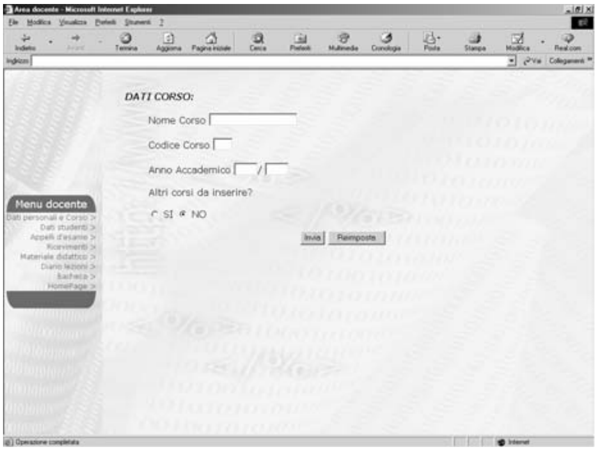

Fig. 7.7. The rendering of the client page `AddCourse.html`

Table 7.4. Valid and not valid sets of values of the input elements in the client page `AddCourse.html`

Input Name	Valid Values	Not Valid Values
Course Code	The course code does not exist in the database	The course code already exists in the database
Course Name	The course name does not exist in the database or it already exists but it is not associated with the same inputted Course Code in the database	The course name already exists in the database and it is associated with the same inputted Course Code
Academic Year	The current academic year	Not Equal to current academic year
More Courses?	{Yes, No}	Not in {Yes, No}
Submit Button	{Clicked, NotClicked}	Not in {Clicked, NotClicked}
Reset Button	{Clicked, NotClicked}	Not in {Clicked, NotClicked}

Page testing was carried out with the aim of verifying that the page was correctly visualised in the browser, and that validated data were sent correctly to the server page AddCourse.asp. To test the page, a driver module allowing the client page to be loaded into a browser, after a registered teacher's login, had to be developed as well as a stub module simulating the execution of the AddCourse.asp server page. This stub module just had to verify that received data coincided with data sent by the client page, and notify the result by a message sent to the client.

Table 7.5. Decision Table for testing the client page `AddCourse.html`

#	Input Section						Output Section	
	Course Name	Course Code	More Courses?	Academic Year First field	Submit	Reset	Expected results	Expected output actions
1	DC	DC	DC	Not Valid	DC	Not Clicked	Academic Year Error Message	Academic year second field filled with the right value.
2	DC	DC	DC	Valid	Clicked	Not Clicked	Data submitted to server page AddCourse.asp Stub notification message about submission correctness	Academic year second field filled with the right value
3	DC	DC	DC	DC	Not Clicked	Click	All the fields in the form have 'blank' values	The page AddCourse.html is Visualized

Note: DC = Don't Care

A set of test cases was defined to exercise all the variants in Table 7.5. These test cases are not reported due to lack of space. The execution of the test cases revealed a failure in the validity check for the AcademicYear values. Indeed, also the value of the Academic Year successive to the current one is accepted (e.g. 2004/2005 is also the valid value of the current Academic Year, and the value 2005/2006 is accepted as valid, while all other successive values, such as 2006/2007, are correctly refused).

This page was also submitted for white box testing. The AddCourse.html page includes HTML code for visualising and managing data input, and two JavaScript functions for validating the Academic Year value. The branch coverage criterion was used to test this page, and all the branches were covered by the set of test cases (the page was instrumented to collect data and verify the coverage). A test model similar to the control flow model proposed by Ricca and Tonella [19] was used to model the structure of the AddCourse.html page and design the test cases.

As an example of server page testing, we consider the AddCourse.asp server page. The responsibility of this page is to register the new course in the database and a course to its teacher, when input values are valid.

To test a server page its code has to be analysed to identify input variables. The input variables found in AddCourse.asp were the ones submitted by the AddCourse.html page (i.e. Course Code, Course Name, More Courses?, Academic Year variables), besides the session variables LoginOK and TeacherCode. Domain analysis was carried out to define valid and invalid sets of values of the input elements. As for the session variable TeacherCode, its valid values were all the ones corresponding to registered teachers and stored in the database. The valid value for the variable LoginOK was the logical value TRUE associated with the condition of an authorized user who made a successful login.

Table 7.6. Decision table for testing the server page `AddCourse.asp`

#	Input section					Output section		
	Course Name	Course Code	More Courses?	Academic Year	LoginOK	Expected results	Expected output actions	Expected state after test
1	Valid	Valid	NO	Valid	True	Data registered into the data base. Success Message	The page TeacherArea.html is visualised	The new course and its association to the teacher has been added to database
2	Valid	Valid	YES	Valid	True	Data registered in the data base	AddCourse.html page is visualised	The new course and its association to the teacher has been added to database
3	DC	Not Valid	DC	DC	True	Error Message	The page CourseMenu.html is visualised	It coincides with the before test state
4	DC	DC	DC	DC	False	Error Message	A new login is required	Coincides with the before test state
5	Not Valid	Valid	DC	DC	True	Error Message	None	Coincides with the before test state
6	DC	DC	Not Valid	DC	True	Error Message	None	Coincides with the before test state

DC – Does not care

To compile the decision table associated with the page, expected results and actions were also looked for. Table 7.6 shows the set of variants we used to test the AddCourse.asp page. The TeacherCode variable was not included in the table because it did not affect the page's behaviour.

As for the testing of the page, a set of test cases was defined in order to exercise each variant for both true and false values. To execute the page test, a driver simulating the HTTP requests by a client, as well as a stub simulating the client pages returned to the client, were developed. No stub was used to simulate the connection to the database, but a test database was used. The execution of this functional testing did not reveal any failure. The same page was submitted for white box testing, using the linearly independent paths coverage criterion. The page included four linearly independent paths which were all covered by the test cases we designed.

At the integration testing level, these two pages were combined and retested together. Table 7.7 reports the decision table including the set of variants used for integration testing. No more failures were observed during the integration testing.

Unit and integration testing were executed for each group of pages implementing the remaining Web application use cases. Thanks to the unit testing, a few failures in some pages were observed. They were mostly due to defects in the validation of user input data. Moreover, we observed that some client pages included JavaScript functions that were not activated during the execution, because they were dead code left in the page after a maintenance intervention that replaced them with new (correctly activated) functions. A similar datum was observed in a few server pages too. As for the integration testing, no additional failure was observed.

As for the unit testing of dynamically generated Web pages, an additional effort was required to design test cases of the server pages responsible for building them. These test cases had to be run, so the client pages were generated, and therefore were captured and stored on-the-fly, to be successively tested. In some cases, the user-session-based approach was exploited to identify test cases able to generate and cover the built client pages.

Table 7.7. Decision table for integration testing of the client page AddCourse.html and server page `AddCourse.asp`

#	Input Section							Output Section	
	Course Name	Course Code	More Courses?	Academic Year	Submit	Reset	Expected results	Expected output actions	Expected state after test
1	DC	DC	DC	Not Valid	Clicked	Not Clicked	Error Message	The page Add-Course.html is visualised	The same as before test
2	DC	Not Valid	DC	DC	Clicked	Not Clicked	Data submitted to server page AddCourse.asp Error Message	The page Corse-Menu.html is Visualised	The same as before test
3	Valid	Valid	NO	Valid	Clicked	Not-Clicked	Data submitted to server page AddCourse.asp. Success Message	The page Teacher-Area.html is visualized	The new course and its association to the teacher added to database
4	Valid	Valid	YES	Valid	Clicked	Not-Clicked	Data submitted to server page AddCorse.asp. Success Message	The page AddCourse.html is visualized again	The new course and its association to the teacher added to database
5	DC	DC	DC	DC	Not-Clicked	Clicked	All the fields in the form put to blank	The page AddCourse.html is visualized	

DC – Does not care

We also executed a system test aimed at exercising each use case implemented by the application at least once. This testing did not reveal any other failures. Moreover, the page coverage reached by this testing was evaluated. All static pages were covered, except one server page which was unreachable, since it was an older version of a page replaced in a maintenance operation.

In conclusion, the experience of functional testing of the "Course Management" Web application was successfully accomplished. Indeed, the fact that testing revealed just a few failures in the application (most of which were due to incorrectly executed maintenance interventions) could be attributed to the "maturity" level of the Web application, which had been running for two years.

The testing experience also highlighted that a considerable effort was required to reconstruct the design documentation needed for test design and execution. This effort might have been saved, or reduced, if this documentation had already been available before testing. This datum can be considered as a strong similarity point between functional testing of a Web application and functional testing of a "traditional" system.

7.8 Conclusions

The openness of Web applications to plenty of users and the strategic value of the services they offer oblige us to consider seriously the verification of both non-functional and functional requirements of a Web application.
While new and specific approaches must be necessarily used for the verification of non-functional requirements (see the problems of security or accessibility testing that are specific for Web applications), most of the knowledge and expertise in the field of traditional application testing may be reused for testing the functional requirements of a Web application.

In this chapter we have reported the main differences and points of similarity between testing a Web application and testing a traditional software application. We considered testing of the functional requirements with respect to four main aspects, i.e. testing scopes, test models, test strategies and testing tools. The main contributions to these topics presented in the literature have been taken into account to carry out this analysis.

The main conclusion we can draw from this discussion is that all testing aspects that are directly dependent on the implementation technologies (such as test models, testing scopes, white box testing strategies) have to be deeply adapted to the heterogeneous and "dynamic" nature of the Web applications, while other aspects (such as black box strategies, or the objectives of testing tools) may be reused with a reduced adaptation effort. This finding also indicates that further research efforts should be spent to define and assess the effectiveness of testing models, methods, techniques and tools that combine traditional testing approaches with new and specific ones.

A relevant issue for future work may be the definition of methods and techniques for improving the effectiveness and efficiency of a Web application testing process. As an example, the adequacy of mutation testing techniques for the automatic validation of test suites should be investigated, as well as the effectiveness of statistical testing techniques in reducing testing effort by focusing on those parts of a Web application that are most frequently used by massive user populations [15]. Moreover, the possibility of combining genetic algorithms with user session data for reducing the costs of test case generation may be a further research question to be investigated. Finally, in the renewed scenario of Web services, new research challenges are being provided by the necessity to consider testing of Web services too.

References

1 Andrews AA, Offutt J, Alexander RT (2005) Testing Web Applications by Modeling with FSMs. Software Systems and Modeling, 4(2)

2 Bangio A, Ceri S, Fraternali P (2000) Web Modeling Language (WebML): a Modelling Language for Designing Web Sites. In: Proceedings of the 9th International Conference on the WWW (WWW9). Elsevier: Amsterdam, Holland, pp 137–157

3 Binder RV (1999) Testing Object-Oriented Systems. Models, Patterns, and Tools. Addison-Wesley: Reading, MA

4 Conallen J. (1999) Building Web Applications with UML. Addison-Wesley: Reading, MA

5 Di Lucca GA, Fasolino AR, De Carlini U, Pace F, Tramontana P (2002) Comprehending Web Applications by a Clustering Based Approach. In: Proceedings of 10th Workshop on Program Comprehension. IEEE Computer Society Press: Los Alamitos, CA, pp 261–270

6 Di Lucca GA, Fasolino AR, Faralli F, De Carlini U (2002) Testing Web Applications. In: Proceedings of International Conference on Software Maintenance. IEEE Computer Society Press: Los Alamitos, CA, pp 310–319

7 Di Lucca GA, Fasolino AR, Tramontana P (2004) Reverse Engineering Web Applications: the WARE Approach. Software Maintenance and Evolution: Research and Practice. John Wiley and Sons Ltd., 16:71–101

8 Elbaum S, Karre S, Rothermel G (2003) Improving Web Application Testing with User Session Data. In: Proceedings of International Conference on Software Engineering, IEEE Computer Society Press: Los Alamitos, CA, pp 49–59

9 Elbaum S, Rothermel G, Karre S, Fisher M (2005) Leveraging User-Session Data to support Web Application Testing. IEEE Transactions on Software Engineering, 31(3):187–202

10 Gomez J, Canchero C, Pastor O (2001) Conceptual Modeling of Device-Independent Web Applications. IEEE Multimedia, 8(2):26–39

11 Harrold MJ, Gupta R, Soffa ML (1993) A Methodology for Controlling the Size of a Test Suite. ACM Transactions on Software Engineering and Methodology, 2(3):270–285

12 Hieatt E, Mee R (2002) Going Faster: Testing The Web Application. IEEE Software, 19(2):60–65

13 Hower R (2005) Web Site Test Tools and Site Management Tools. Software QA and Testing Resource Center. www.softwareqatest.com/qatWeb1.html (accessed 5 June 2005)

14 IEEE Std. 610.12–1990 (1990). Glossary of Software Engineering Terminology, in Software Engineering Standard Collection, IEEE Computer Society Press, Los. Alamitos, CA

15 Isakowitz T, Kamis A, Koufaris M (1997) Extending the Capabilities of RMM: Russian Dolls and Hypertext. In: Proceedings of 30th Hawaii International Conference on System Science, Maui, HI, (6):177–186

16 Kallepalli C, Tian J (2001) Measuring and Modeling Usage and Reliability for Statistical Web Testing. IEEE Transactions on Software Engineering, 27(11):1023–1036

17 Liu C, Kung DC, Hsia P, Hsu C (2000) Object-based Data Flow Testing of Web Applications. In: Proceedings of First Asia-Pacific Conference on Quality Software. IEEE Computer Society Press, Los Alamitos, CA, pp 7–16

18 Nguyen HQ (2000) Testing Applications on the Web: Test Planning for Internet-Based Systems. John Wiley & Sons, NY

19 Ricca F, Tonella P (2001) Analysis and Testing of Web Applications. In: Proceedings of ICSE 2001 IEEE Computer Society Press, Los Alamitos CA, pp 25–34

20 Ricca F, Tonella P (2004) A 2-Layer Model for the White-Box Testing of Web Applications. In: Proceedings of Sixth IEEE Workshop on Web Site Evolution IEEE Computer Society Press, Los Alamitos, CA, pp 11–19

21 Sampath S, Mihaylov V, Souter A, Pollock L (2004) A Scalable approach to user-session based testing of Web Applications Through Concept Analysis. In: Proceedings of 19th International Conference on Automated Software Engineering, IEEE Computer Society Press: Los Alamitos, CA, pp 132–141

22 Sampath S, Mihaylov V, Souter A, Pollock L (2004) Composing a framework to automate testing of operational Web-based software. In: Proceedings of 20th International Conference on Software Maintenance IEEE Computer Society Press pp 104–113

23 Schwabe D, Guimaraes RM, Rossi G (2002) Cohesive Design of Personalized Web Applications. IEEE Internet Computing. 6(2):34–43

24 Web Content Accessibility Guidelines 2.0 (2005), http://www.w3.org/TR/WCAG20 (accessed 5 June 2005)

Authors’ Biographies

Giuseppe A. Di Lucca received the Laurea degree in Electronic Engineering from the University of Naples “Federico II”, Italy, in 1987 and the PhD degree in Electronic Engineering and Computer Science from the same university in 1992.

He is currently an Associate Professor of Computer Science at the Department of “Ingegneria” of the University of Sannio. Previously, he was with the Department of 'Informatica e Sistemistica' at the University of Naples “Federico II”. Since 1987 he has been a researcher in the field of software engineering and his list of publications contains more than 50 papers published in journals and conference proceedings.

He serves on the programme and organising committees of conferences in the field of software maintenance and program comprehension. His research interests include software engineering, software maintenance, reverse engineering, software reuse, software reengineering, Web engineering and software migration.

Anna Rita Fasolino received the Laurea degree in Electronic Engineering (cum laude) in 1992 and the PhD degree in Electronic Engineering and Computer Science in 1996 from the University of Naples “Federico II”, Italy, where she is currently an Associate Professor of Computer Science. From 1998 to 1999 she was at the Computer Science Department of the University of Bari, Italy.

Her research interests include software maintenance and quality, reverse engineering, Web engineering, software testing and reuse, and she has published several papers in journals and conference proceedings on these topics. She is a member of programme committees of conferences in the field of software maintenance and evolution.