Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione (DINFO)
PhD Course in Information Engineering

Curriculum: Computer Engineering
Academic Discipline (SSD): ING-INF/05

Verification of Enterprise Software Architectures with stateful managed components

Candidate
Dott. Jacopo Parri

Supervisors
Prof. Enrico Vicario

Prof. Alessandro Fantechi

PhD Coordinator
Prof. Fabio Schoen

---

cycle XXXIII, 2017-2020

Università degli Studi di Firenze, Dipartimento di Ingegneria
dell'Informazione (DINFO).

Thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Information Engineering. Copyright © 2021 by
Dott. Jacopo Parri.

Acknowledgments

I would like to acknowledge the efforts and input of my scientific supervisors, Prof. Enrico Vicario and Prof. Alessandro Fantechi, and all my colleagues of the *Software Technologies Lab* (STLAB), within the Information Engineering Department (DINFO), who were of great help during the PhD. In particular my thanks goes to Dott. Samuele Sampietro, trusted friend of "pair programming", who completely shared with me this PhD experience, also collaborating on the main parts of this research.

Abstract

In the engineering of software applications designed with modular Enterprise Architectures, the management of components and their dependencies is often delegated to an outer participant, named *container*, which assumes the primary responsibility of taking care of creation, destruction and dependencies resolution of all managed software components, realising the so called Inversion of Control (IoC) principle.

This dissertation contributes to the area of Model-Based Testing, proposing a methodology for verification of Enterprise Software Architectures with *stateful* components, exploiting Dependency Injection (DI) and automated contexts management.

The research addresses the problem of test case generation for Web Applications, implementing the IoC principle through the adoption of *state of the art* DI containers and frameworks with contexts management capabilities and with built-in contexts, defined according to client-server paradigm and HTTP fundamentals.

At the core of the methodology a new abstraction, named Managed Components Data Flow Graph (*mcDFG*) is proposed for supporting the test case generation stage, addressing a fault model which identifies specific types of fault affecting *stateful* applications. The *mcDFG* reinterprets classical Data Flow Graph theory, combining structural information with navigational and behavioural aspects of component-based applications. A set of coverage criteria, applied over the *mcDFG*, supports the automated extraction of paths, each one representing a reference description of a single test case, prescribing the sequence of end-user interactions which must be implemented to exercise the System Under Test in an *end-to-end* testing perspective.

The proposed methodology is also integrated with consolidated practices of software development so as to leverage on common design and documentary artefacts for enabling the automated generation of the *mcDFG*.

An evaluation of the applicability of the methodology, in support of its convenience, is discussed for a prototype Web Application, implemented with the Java™ Enterprise Edition ecosystem through the Contexts and Dependency Injection (CDI) specification as the DI container, highlighting capabilities in the generation of an effective test suite for the characterised fault model.

Contents

Contents v

1 Introduction 1

1.1 Contributions . . . 4

2 Enterprise Architectures and Inversion of Control 7

2.1 Enterprise Software Architectures . . . 8

2.2 Inversion of Control . . . 10

2.3 Automated Contexts Management . . . 12

2.4 Technologies for Dependency Injection . . . 14

2.4.1 Contexts classification for concrete DI frameworks . . . 15

3 Literature review 21

3.1 Abstractions for modelling component-based applications . . . 22

3.1.1 Structural design of components . . . 22

3.1.2 Navigational and behavioural design . . . 26

3.2 Model-Based Testing . . . 29

3.2.1 Structural Testing . . . 30

3.2.2 Functional Testing . . . 31

3.3 Non-Model-Based approaches . . . 34

4 Running Case Study 37

4.1 Operative context . . . 38

4.2 Functional design . . . 40

4.3 Architecture . . . 41

4.4 Navigation design . . . 46

5 Motivations and Problem Formulation **49**
5.1 Motivations 50
5.2 Problem Formulation 54
5.3 Fault Model 57
5.3.1 Fault Model conceptualisation 59
5.3.2 Fault Model concretisation 60

6 Verification of stateful Web Applications **73**
6.1 Preamble 74
6.1.1 The mcDFG abstraction 74
6.2 The Methodology 76
6.2.1 Structural and behavioural preliminary analyses . . . 76
6.2.2 Robustness analysis 77
6.2.3 Robustness Diagram decoration 80
6.2.4 Managed Components DFG generation 82
6.2.5 Test Case generation 89
6.2.6 Summary 94

7 Discussion **97**
7.1 Evaluation of the methodology 98
7.1.1 Fault hunting within the Case Study 98
7.2 Final discussion 106

8 Conclusion **109**
8.1 Summary of contributions 109
8.2 Directions for future work 111

A Appendix **113**
A.1 *mcDFG* generation algorithms 114
A.2 Further artefacts and abstractions 117

B Publications **121**

Bibliography **123**

Chapter 1

Introduction

In the engineering of a wide class of Web Applications, especially in the construction of modular Enterprise Software Architectures, the management of instantiated components, and their dependencies, becomes a crucial element of the overall application complexity. To mitigate it in a productive way, the control of components lifecycle and the runtime installation of required dependencies are often delegated to an outer participant, named *container*. In so doing, the container assumes the primary responsibility of taking care of creation, destruction and dependencies resolution of all managed software components, realising the so called Inversion of Control (IoC) principle.

A common and practical implementation of the IoC principle, in many high-level programming languages, is often provided by advanced frameworks which include *ad hoc* containers, complying with the Dependency Injection (DI) mechanism and automated contexts management capabilities. These solutions promote loose coupling and support application designers and developers automatically binding components lifecycle to built-in contexts and scopes, thus defining and delimiting visibility boundaries as well as controlling related runtime object instances (i.e., the so-called contextual instances) through adequate construction and destruction policies.

Although a DI container may solve, in background, runtime dependencies and automatically manage contextual instances, so relieving developers from this burden, the overall complexity of the implementation grows. In these cases, dealing with the testing stage may become more difficult and exhaust-

ing, also considering that the realised decoupling between components definitions (e.g., directly allowing meta-configurations within the source code) and components instances slightly blurs the developer's overview about the overall application structure.

A lack of design control is unavoidable in medium-large size software applications, thus increasing the complexity in the definition of an effective test suite.

Under the above premises, the research activity described in this dissertation addresses the formulation of a verification methodology aimed at recognising potential faults within Web Applications designed and developed with a *stateful* behaviour, controlled by a DI container and constrained to pattern-oriented design approaches. The methodology also represents a guideline for supporting designers, developers, and testing specialists in the generation of effective test suites in end-users scenarios where several managed components cooperate in realising a common use case.

The problem is exacerbated by the intrinsic nature of Web Applications which is conditioned by the client-server paradigm, the HTTP protocol and the interpretation given to the concepts of *state*, components *scope* and *visibility*: during end-users interactions, the server-side allocates some runtime object instances, populates the visited view pages, and stores in-memory a set of managed components with their dependencies.

Placing the state on the server-side, increases the server control over the application behaviour; but, at the same time, the ability to control and detect cases of wrong lifecycle management, unexpected end-user interactions, or hidden dependencies is reduced, raising also the presence of potential implementation defects, which could activate latent faults.

During the research, a review among widespread DI frameworks for main programming languages (i.e., C#, Java™, and Python™) has been accomplished with the aim of understanding how they differently interpret the IoC principle, providing several and specific scopes for managed components. The review lays the foundation for a classification, in a general perspective, of the available contexts within DI containers, thus offering a common vision on *state of the art* technologies in modern *stateful* architectures.

An *ad hoc* fault model, accounting scenarios of collaborative components

has been characterised, identifying four types of fault, tailored on Web Application scenarios which may produce unexpected cases of memory leakages, data inconsistencies, race conditions, behavioural ambiguities or other kind of failures.

The proposed methodology has been formalised as an artefact-driven approach, in a Model-Based Testing perspective, integrated in consolidated practices for software development, exploiting preliminary stages of software requirements specification, use cases design, and robustness analysis.

The methodology, inspired by classical Data Flow Testing (DFT) theory, supports the test case generation process, extracting significant paths over a new abstraction, named Managed Components Data Flow Graph (*mcDFG*), which extends the classical Data Flow Graph (DFG) with redefined concepts of *defs* and *uses* for component-based applications subject to DI as well as including navigation information and end-user interactions. Also main coverage criteria for the *mcDFG* have been redefined with respect to classical ones for a DFG.

Finally, the methodology has been applied over an *Online Flight Booking IT System*, developed in Java™ Enterprise Edition, offering a basis for the discussion about how the methodology is able to support the detection of faults identified within the proposed fault model through a test case generation stage applied over related *mcDFG* artefacts.

The rest of the dissertation is organised as follows: in **Chapter 2**, fundamental concepts about Enterprise Software Architectures and IoC principle are provided. Furthermore, a comparative review of main technologies for DI is addressed with a final classification of built-in contexts provided by reviewed DI containers; in **Chapter 3**, the literature review about Model-Based Testing techniques, both in structural and functional perspectives, is presented together with structural and navigational abstractions for component-based applications; in **Chapter 4**, a description of the main technical aspects (e.g., required design artefacts and architectural overview) of the running case study is introduced for facilitating the presentation and argumentation of the subsequent chapters; in **Chapter 5**, the motivations and a problem formulation about testing of *stateful* component-based applications, exploiting DI with automated contexts management, are addressed; besides, a conceptualisation of a fault model as well as its concretisation within the running

case study are provided; in **Chapter 6**, the methodology for verification of *stateful* Web Applications is presented, with a special focus on the definition of the core abstraction, its construction process and its adoption in test case generation stages with related coverage criteria; in **Chapter 7**, a discussion about the applicability of the proposed methodology over the most significant use cases of the running case study is provided, highlighting fault detection capabilities for each fault type concretisation of the identified fault model; finally, conclusions and future research plans are drawn in **Chapter 8**.

1.1 Contributions

The research described in this dissertation proposes a methodology for verification of Enterprise Software Architectures with *stateful* components, exploiting Dependency Injection (DI) and automated contexts management, thus contributing to the area of Model-Based Testing.

The main contributions are here summarised:

- an artefact-driven methodology for test case generation tailored for pattern-oriented Web Applications with *stateful* behaviour controlled by a DI container providing automated contexts management;

- a review of DI frameworks in the stack of major programming languages (i.e., C#, Java™, Python™) comparing supported types of context within which the components live and are managed by the DI container;

- a characterisation of the fault model affecting applications with managed components, leading to the identification of four specific types of fault: vanishing components, zombie components, unexpected shared components, and unexpected injected components;

- the definition of an abstraction that addresses the fault model by combining a structural perspective, related to dependencies among components, with a navigational and behavioural perspective, related to end-users interactions. The abstraction, named *mcDFG* is based on the classical Data Flow Graph, reinterpreted in the meaning of *defs* and *uses* and enriched with salient characteristics of *stateful* Web Ap-

plications (e.g., navigation actions, component injections, method invocations, contexts management);

- a procedure for building the *mcDFG* abstraction starting from an enriched version of the UML Robustness Diagram, opening the way to automate the *mcDFG* construction, thus integrating the whole methodology with consolidated practices of software development;
- the identification of *ad hoc* coverage criteria, based on an implementation of the concepts of Data Flow Testing theory, adapted to the characteristics of the proposed *mcDFG*, re-evaluating inclusion relationships among them;
- a qualitative discussion about the applicability of the methodology on a middle-size application, implemented with the Java™ Enterprise Edition technological stack including Contexts and Dependency Injection (CDI) specification as DI container, highlighting capabilities in the generation of an effective test suite.

Chapter 2

Enterprise Architectures and Inversion of Control

In this Chapter a brief overview of main Enterprise Software Architectures is provided, discussing also the adoption of Inversion of Control principle for automatically manage components dependencies and components lifecycles, focusing on Web Applications designed with a stateful behaviour.

A comparison of state of the art architectural styles for enterprise solutions is reported in Sect. 2.1, highlighting the dichotomy between stateless and stateful applications.
In Sect. 2.2, the fundamentals of Inversion of Control with a specific focus on Dependency Injection (DI) are described; while in Sect. 2.3, mechanisms for automated contexts management are presented with the aim of identifying salient traits of concrete production frameworks, also proposing a conceptual classification of common contexts, compared with main DI technologies for widely adopted programming languages in Sect. 2.4.

2.1 Enterprise Software Architectures

Enterprise Architectures [97] have been widely advocated in the design and implementation of complex and distributed information systems within enterprise and industrial areas, for realising software applications able of dynamically adapting to business models and internal processes, keeping aligned the evolution of Information Technology (IT) systems with corporate missions and needs, thus facilitating maintenance, decision making, and planning operations.

Fundamental principles of Enterprise Software Architectures notably include concepts of reusability, flexibility, agility, and efficiency, leading to the adoption of the so called component-based applications, leveraging on technological and functional decoupling among distinct "blocks" of software to be used without implementation changes on their source code, as stated by the Open Closed Principle [73]. Under this assumption, software components may be directly bundled within client applications or may be remotely invoked through specific communication standards and protocols.

In so doing, an Enterprise Software Architecture refers to a family of architectural styles, designed in such a way as to ensure a weighted and evolutionary growth of organisations' IT systems, both in functional and qualitative terms.

In the field of Software Architectures, especially for Web Applications built over HyperText Transfer Protocol (HTTP) [10, 33], two main architectural styles have become widespread and mature:

- monolithic architectures exploiting software components which are strictly constrained to contexts residing on the server-side. These architectures are usually characterised by a *stateful* behaviour with strongly coupled backend and frontend modules;

- service-oriented architectures (SOA) exploiting software components exposing a business logic through services or remote procedures. Emerging paradigms for these architectures are usually characterised by a *stateless* behaviour with decoupled backend and frontend modules.

On the one hand, many applications monolithic-based are designed as *N-tier* architectures [71], also known as *N-layer* architectures, often, requiring at least *3-tiers*: *i)* a presentation layer exposing Graphical User Interfaces for

end-users, also providing predefined events on user interactions; *ii)* a business logic layer, concretely implementing exposed functionalities in reaction to user-events, and *iii)* a data-access layer exploiting object-relational mappers [68] so as to guarantee compliance between domain model instances and data schemes defined within underlying Relational Database Management Systems. This behaviour is typical of applications developed under the *Model-View-Controller* [25] architectural pattern and, in the specific case of *stateful* applications each view is built and populated by the server-side, and then transferred to the requesting clients.

The functional stratification promotes the *separation of concerns* and *information hiding* principles [29, 81, 82], while maintaining a strong coupling among the components responsible of organising and populating the presentation tier (i.e., the views of the frontend module) and the components responsible of implementing the business logic for each designed application use case (i.e., the controllers of the backend module).

On the other hand, application SOA-based are designed so as to cope with the concept of *service* (i.e., a self-contained and atomic activity to be executed in the business logic) and with the aim of encouraging *Enterprise Application Integration* among heterogeneous applications. Under this perspective, clients interpret the application as a kind of *black box*, exposing and exchanging resources under a defined data contract, a common transfer interface which fixes the request parameters and the response data format (e.g., HyperText Markup Language, eXtensible Markup Language or JavaScript Object Notation).

Among SOA-based applications, a relevant architectural style is the REpresentational State Transfer (REST) [34] which promotes the *resource* [95] as a key concept: a server static or dynamic asset (e.g., a service, a document, an image, a set of multiple of them) that is possible to find and retrieve through its unique identifier.

The fundamental principles of RESTful architectures entail a shift of responsibilities, roles, and functionalities between a service layer, exposed by the backend, and the business logic of a variety of clients with different needs and purposes (e.g., a classical online application via web browser versus a native mobile app) which may consume services in a variety of usage scenarios through decoupled frontends. These principles prescribe the adoption of HTTP as the transport protocol and the exploitation of the deep seman-

tics of its verbs (e.g., POST, GET, PUT, DELETE) in CRUD operations (i.e., Create, Retrieve, Update, and Delete), as well as the prescription of a *stateless* behaviour (i.e., each request made by any client must therefore contains all the information necessary for the server to generate a response). [96]

This dissertation, proposing a verification methodology for component-based *stateful* applications, directly addresses the case of monolithic architectures, which have been widely adopted for realising enterprise information systems, inherited by legacy systems or developed within contexts where security, at data-level, comprises a core requirement (e.g., bank transactions). Indeed, the retention of data and contexts on the server-side simplifies the definition and the verification of authorisation policies with respect to the case of *stateless* architectures, where the authorisation must be replicated by the backend module at each received request.

In a wider perspective, both migration from legacy systems to SOA applications and the definition of methodologies to handle server-side data for security constraints, may lead to the design of "hybrid" architectures exploiting a *stateful* behaviour over SOA applications based on RESTful principles.

2.2 Inversion of Control

In the development practice of component-based Object-Oriented (OO) software architectures, the Inversion of Control (IoC) principle [36, 38] has been widely adopted so as to automate components instantiation and dependencies management, thus promoting loose coupling: in this way, software components are relieved of the responsibility of installing references, automatically resolving all the dependencies at runtime.

The etymology of IoC can be traced back to the paper of Johnson and Foote [51], which literally says: "methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code".

The authors' words can be interpreted as follows: a programming language or library respecting the IoC principle should provide to the developer a *framework* to automatically perform a set of specific actions.

Following the consolidated definition stated by the Gang of Four in [41],

a framework is a collection of cooperative classes providing a reusable skeleton useful to design and define the architecture and the general structure of the application where is adopted, thus realising the IoC principle. In fact, a significant characteristic of a framework consists in its ability of pre-defining the general responsibilities and the interaction modalities among classes and objects, also controlling and orchestrating activities (e.g., object instantations and control flow) rather than delegate them to developers which can therefore focus on specific choices related to application business logic.

From this perspective, the framework offers a set of general purpose solutions to developers which must tailor them in order to customise the final application behaviour.

For all above considerations, it becomes quite clear the reason why the IoC principle is also known as the *Hollywood Principle*, introduced by Sweet and metaphorically explained in [111] as "Don't call us, we'll call you (Hollywood's Law). A tool should arrange for Tajo[1] to notify it when the user wishes to communicate some event to the tool, rather than adopt an '*ask the user for a command and execute it*' model.".

In Software Engineering, various implementations of the IoC principle have been proposed, notably including Service Locator [2] and Dependency Injection [89] patterns.

On the one hand, the basic idea behind the Service Locator, is the definition of a *singleton component*, containing basic implementations for each instantiable component, in a kind of factory method approach. In so doing, the singleton instance acts as a central registry with all the responsibilities about performing lookup of distributed services and their creations.

On the other hand, the Dependency Injection (DI) is based on the design of an automated *container*, also known as *assembler*, with the responsibility of constructing in background the runtime components, choosing the right class/type, and installing dependencies (which, in turn, must be instantiated and whose dependencies must be resolved). In so doing, the DI is a form of the Dependency Inversion Principle (DIP) [72] which states that components should depend upon abstractions and not on concretions [73].

[1] *Tajo* is the code-name of the User Interface of the Mesa project, a programming environment described within the paper [111].

This dissertation primarily focuses on frameworks based on DI, which are widespread in many *state of the art* programming languages (as described in Sect. 2.4), but main considerations may be adapted so as to cope with the Service Locator pattern.

2.3 Automated Contexts Management

Many practical implementations of the DI mechanism are commonly accompanied by automated control of the lifetime and the visibility of injected components, further promoting decoupling by assigning to the DI container, provided by the framework, the responsibility of creating, sharing, and destroying managed objects.

In so doing, resolved dependencies (i.e., objects injected in dependent components) are constrained to conceptual boundaries, delimiting components lifecycle and admitted interactions within the *stateful* application operative domain: these digital confined areas are named *contexts*.

All the objects managed by the DI container, living constrained within available and active contexts, are named *contextual instances* of managed components. In many cases, references are intermediated by runtime *proxies*, decoupling the container control from their concrete contextual instances.

Contexts may take on different meanings depending on the way a DI framework interprets them or on the way a software application has been designed and distributed to the final end-users (e.g., desktop standalone versus online services).

This dissertation focuses on Web Applications leveraging on the client-server paradigm and exploiting inter-connectivity provided by Internet so as to remotely offer functionalities, which may be enjoyed ubiquitously and within different platforms. Most Web Applications adopt HTTP as the core network protocol, guaranteeing the delivery throughout its underlying protocols (i.e., Transmission Control Protocol and Internet Protocol), for packaging User Interfaces written as HyperText Markup Language (HTML) documents, supported by Cascading Style Sheets (CSS) for styling web pages and JavaScript for client-side scripting.

In so doing, DI frameworks, specifically addressing the case of *stateful* Web Applications, are based on a conceptualisation of contexts strictly defined over the HTTP protocol and its basic features (e.g., request, session).

Despite HTTP is a *stateless* protocol and does not natively allow information sharing among different HTTP requests for longer living data, through the HTTP State Management Mechanism [7] the protocol overcomes these limitations, enabling servers to store data along user sessions. Thanks to this feature, developers are enabled in implementing applications with a short-term memory maintained along use cases (e.g., exploiting server RAMs to allocate runtime data without the constraint of persisting them into a long-term database), thus offering a better experience to the end-users during their interactions.

Taking into account above considerations, a classification and identification of fundamental contexts for a Web Application is here addressed.

With reference to the built-in concepts of the HTTP protocol, two main contexts have been identified:

- the HTTP *request* can be considered as the basic context, representing the minimum communication boundary between client and server, implying that managed components with this scope are allocated and maintained server-side only for the necessary time to generate a single response;
- the HTTP *session* can be considered as the context including data retention produced through the HTTP requests spanned from the "login" use case, which allocates data after identification and authentication processes, to the "logout" use case, which releases server-side per user memory. Session data are stored in each HTTP request and accessed by the server through a unique session identifier provided by the client, often specified inside *HTTP Cookie* and *Set-Cookie* header fields or inside a query parameter or within the *HTTP Authorization* header (e.g., adopting username and password credentials or JSON Web Tokens [54]).

Instead, with reference to typical aspects of a software application, other two contexts can be considered:

- the whole application lifespan can be embedded within an *application* context, including component contextual instances conceptually equivalent to singleton instances with a global visibility and not tied to single user sessions;

- some data may be shared within a *use case* context, spanned among several HTTP requests performing an atomic end-user unit of work, whose boundaries should be manually managed by developers. As with the *session* context, a *use case* context is associated with a unique identifier exchanged between client and server.

By definition, all the above mentioned contexts are organised in a hierarchical fashion: *application* context wraps *session* contexts, which in turn, wrap several *use case* contexts, which are composed by a set of *request* contexts.

Many DI frameworks also provide a kind of *inherited* context, named *prototype*, which consists in the injection of a different contextual instance for each dependent component, binding moreover the lifecycle of the injected instance to the lifecycle of the dependent one. The *prototype* context, for a managed component, can be interpreted as a "pseudo-scope".

Besides, many DI frameworks expose Application Programming Interfaces or Service Provider Interfaces for enabling the programmatic definition of custom contexts and their behaviours; so further built-in contexts exploiting specific and advanced feature of Web Applications (e.g., WebSocket) may be disposed. These types of contexts, being less commons and less standardised, are out of scope for this dissertation.

2.4 Technologies for Dependency Injection

Inversion of Control becomes effective in concrete DI frameworks implementations provided by several programming languages (e.g., C#, Java™, Python™), exploiting different approaches and perspectives driven by architectural intents. Indeed, DI is a key principle for many backend (e.g., CDI specification in Java™) and frontend (e.g., Angular DI in TypeScript) frameworks.

On the one hand, backend frameworks aim at supporting server-side state management and data sharing within *stateful* applications, especially for those based on monolithic architectures, widely exploiting web contexts.

On the other hand, frontend frameworks aim at supporting client-side automated components injections within the User Interfaces which are decoupled by the state management and the data sharing, usually leveraging on user-agents data storages (e.g., session and local storages of web browsers) instead of relying on automated contexts management.

In the following Sect. 2.4.1, a review of DI frameworks in majors enterprise level technologies is addressed, focusing on backend frameworks.

2.4.1 Contexts classification for concrete DI frameworks

In this Section, a brief review of main technologies and frameworks for DI and automated contexts management is reported, classifying them on the basis of their reference programming language. At the end of the review, a comparison table is reported highlighting, for each presented DI framework, the contexts (e.g., request, session, application) managed by its DI container.

The review is useful to understand how *state of the art* frameworks interpret the IoC principle and, specifically, which contexts are provided by DI containers, thus becoming interesting for the methodology proposed within this dissertation.

C#

In C#, the *Autofac* [103] framework represents the primary technological solution for .NET, .NET Core, and ASP.NET Core based applications. The framework enables configurations of components through a rich programmatic API, starting from a common instance of builder, invoking the method named *RegisterType<...>()* in chaining with one of the following methods:

- *InstancePerRequest()* binds a component to a single HTTP request. It can be interpreted as a *request* context;
- *SingleInstance()* communicates to the container to instantiate and sharing a single instance during the whole lifetime of an application. It can be interpreted as an *application* context;
- *InstancePerDependency()* indicates to instantiate a single and different instance of a component for dependency. It can be interpreted as a *prototype* context;
- *InstancePerLifetimeScope()* binds a component to a programmatic scope, also in a nested mode. It can be interpreted as a *use case* context, supporting also the special case of the *session* context;
- *InstancePerMatchingLifetimeScope()* binds a component to a named lifetime scope, facilitating the identification of scope boundaries. It

can be interpreted as an alternative way to access *use case* and *session* contexts;

- *InstancePerOwned()* binds a component to a single dependent owner type. It can be interpreted as a *prototype* context;
- *InstancePerThread()* binds a component to a single CPU thread. It represents a custom context.

Spring.NET [87] is another C# open-source application framework, easing web development practices by enabling DI mechanisms with automated contexts management. Components scopes may be declared within XML configuration files (within the *scope* attribute). Built-in scopes include:

- *"request"* providing a single component instance for each HTTP request. It can be interpreted as a *request* context;
- *"session"* providing a single component instance for each HTTP session. It can be interpreted as a *session* context;
- *"application"* providing a single component instance to the entire application lifetime of a Web Application. It can be interpreted as an *application* context;
- *"singleton"* providing a single component instance, as in the case of *"application"*, but for standalone programs. It can be interpreted as an *application* context;
- *"prototype"* providing a different component instance for each dependent component. It can be interpreted as a *prototype* context.

Java™

In Java™, many DI frameworks enable configurations of components bindings through decorations applied directly on classes (i.e., through annotations) and/or with configuration files in eXtensible Markup Language (XML) documents.

Java™ Enterprise Edition (JEE) includes within its core modules the *Contexts and Dependency Injection* (CDI) specification, which is presently implemented by various providers, notably including JBoss Weld [57] and Apache

OpenWebBeans [4]. The specification is defined through a set of Java™ Specification Requests (JSRs), since JSR-299 [58] to JSR-365 [100]. A CDI managed component, also called *bean*, is associated with one of a limited number of built-in scopes, available through following class annotations:

- *@RequestScoped* binds a component to a single HTTP request. It can be interpreted as a *request* context;
- *@ConversationScoped* binds a component to multiple HTTP requests through a conversation id parameter (i.e., *cid*) inside a single HTTP session. Conversation boundaries can be manually defined and managed by the developer. It can be interpreted as a *use case* context;
- *@SessionScoped* binds a component to a single HTTP session. It can be interpreted as a *session* context;
- *@ApplicationScoped* binds a component to the entire application lifetime. It can be interpreted as an *application* context;
- *@Dependent*, a pseudo-scope that binds a component inside a dependent one injecting it. This means that making different injections of the same dependent bean, in the same context, results in multiple not shared contextual instances. It can be interpreted as a *prototype* context.

The Spring Framework provides a custom implementation of DI within the *Spring IoC Container* [53], complying with the JSR-330 [52] specification, enhanced by the definition of built-in contexts:

- *@RequestScope* binds a component to a single HTTP request. It can be interpreted as a *request* context;
- *@SessionScope* binds a component to a single HTTP session. It can be interpreted as a *session* context;
- *@ApplicationScope* binds a component to the entire servlet-container lifetime, sharing a single component also among different applications running on the same server. It can be interpreted as an *application* context;
- *@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)* binds a component to the entire application lifetime. It can be interpreted as an *application* context;

- *@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)* indicates to the container to bind a single and different instance of the component to each dependent one. It can be interpreted as a *prototype* context;
- *@Scope(scopeName = "websocket")* supports unique instantiations of components in each websocket channel. It represents a custom context.

Google provides a lightweight DI framework for Java™ 6+, named *Guice* [115], useful in applications that do not have an intensive use of *stateful* contexts. This framework exposes a limited number of contexts:

- *@RequestScoped* binds a component to a single HTTP request. It can be interpreted as a *request* context;
- *@SessionScoped* binds a component to a single HTTP session. It can be interpreted as a *session* context;
- *@Singleton* communicates to the container to instantiate a single instance during the whole lifetime of an application. It can be interpreted as an *application* context;
- *by default*, in absence of explicit configurations, the container provides a different instance at each injection. This behaviour can be interpreted as a *prototype* context.

Python™

In Python™, for applications written in Object-Oriented perspective, the main DI framework is *Dependency Injector* [64], which facilitates developers in explicitly declaring dependencies and performing components injections. The framework, unlike previous ones for other programming languages, does not offer contexts representations, thus avoiding their automated management. In so doing, each component lives in a *prototype* context.

Another Python™ library for DI, named *Pinject* [43], has been offered by Google with the aim of supporting the assembly of components into graphs, also providing two built-in scopes which control objects memoization strategies (i.e., caching). These scopes are:

- *PROTOTYPE*, which does not allocate objects into a cache. It can be interpreted as a *prototype* context;

- *SINGLETON*, the default scope, which always binds object components to a cache. It can be interpreted as an *application* context.

The *Guice* framework, introduced for Java™, has inspired a third Python™ framework, named *Injector* [112] which provides specific declarative annotations (i.e., *@inject*) for defining inline injection points within the source code, and a special class (i.e., *Injector*) for performing programmatic injections (in the framework terminology, this practice is named *assisted injection*). The *Injector* framework also provides two built-in scopes: *NoScope*, corresponding to an *unscoped* provider interpretable as a *prototype* context, and *@singleton*, corresponding to an *application* context. At the same time, it is possible to define custom scopes, by sub-classing the *Scope* class and defining a custom decorator annotation (e.g., *@custom*), so enabling the definition of different contexts (e.g., *request*, *session*, *use case*).

Summary An interpreted mapping between frameworks built-in contexts with the proposed conceptual classification of Sect. 2.3 is reported in Tab. 2.1.

		request	*use case*	*session*	*application*	*prototype*
C#	Autofac	✓	✓	✓	✓	✓
C#	Spring.NET DI	✓		✓	✓	✓
J	CDI	✓	✓	✓	✓	✓
J	Spring DI	✓		✓	✓	✓
J	Guice	✓		✓	✓	✓
P	Dependency Injector					✓
P	Pinject				✓	✓
P	Injector	✓	✓	✓	✓	✓

Table 2.1: Comparison among available contexts for primary DI frameworks. The first column contains a reference letter to the respective programming language (i.e., C# := C sharp, J := Java™, P := Python™).

Chapter 3

Literature review

*This Chapter gives a brief survey of related works about main abstractions for modelling component-based software applications, in Sect. 3.1, addressing two different perspectives: **i)** structural abstractions, for modelling the internal organisation of a software system in terms of components and existent relationships; and **ii)** navigational and behavioural abstractions, for modelling the internal business processes of a software system, capturing its internal behaviour in terms of exposed functionalities, collaborations among components and interaction sequences driven by use cases.*

The review also includes an overview of Model-Based Testing approaches, in Sect. 3.2, leveraging on formal and semi-formal models as primary documentation artefacts, leading the choice of a stimulus to the System Under Test and its verification. Also in this case, following the selected perspectives of modelling abstractions, the dissertation distinguishes between structural testing (i.e., white box) and functional testing (i.e., black box) approaches.

Finally, for the sake of completeness, a brief review of the main non-model-based approaches is provided in Sect. 3.3.

3.1 Abstractions for modelling component-based applications

In this Section, a brief review of the main abstractions for modelling component-based software systems is reported, focusing in two different perspectives.

On the one hand, abstractions capturing *structural* characteristics of a system enable fine modelling of software components in isolation or in mutual dependence, relying on knowledge extracted from the implementation.

On the other hand, *navigational* and *behavioural* abstractions provide capabilities for reducing the complexity of the verification problem by representing only feasible sequences of operations and system behaviours in accordance also with functional requirements specifications and use cases.

3.1.1 Structural design of components

Many models and abstractions have been proposed in literature for capturing structural characteristics of software programs, evolved in time so as to adapt to emergent programming paradigms (e.g., procedural, Object-Oriented, Aspect-Oriented), applications complexity, and innovative testing techniques; in this subsection, the most significant ones for the research described in the dissertation are mentioned.

In software analysis and design processes, probably, the reference standard for designers and developers of Object-Oriented applications is the *Unified Modeling Language* (UML) [11], defined by OMG (Object Management Group), providing a set of useful and extensible diagrams, supporting several designing stages within different perspectives (i.e., structural or behavioural).

Component-based applications designed from a structural perspective rely on diagrams with a static nature, such as: *Class Diagrams*, for modelling set of classes, entities, and interfaces with existent relationships; *Object Diagrams*, for modelling object instances referred to a subset of classes within a possible runtime scenario; *Package Diagrams*, for modelling packages dependencies; *Component Diagrams* for modelling application components focusing on organisation and dependencies; and *Deployment Diagrams*, for modelling the the deployment architecture considering involved software components, communication interfaces and also hardware items (e.g., Application Servers, Database Management Systems).

Salient characteristics of software components, in Object-Oriented applications, can be captured through UML standard abstractions, but advanced features of Web Applications (e.g., managed contexts derivable from HTTP, dynamic dependencies managed server-side) need for extensions in syntax and semantics. For these purposes, UML notation provides extensibility and customisability of representational graphic elements enabling the definition of *ad hoc* languages for specific domains, through *UML Profiles* [40].

In [21, 22] is introduced and presented the *Web Application Extension* (WAE) UML profile, supporting design activities for Web Applications through the abstraction of *ad hoc* primitives about pages, forms, links, redirects, scripts, and style sheets.

In [109], a UML profile named *FrameWeb* is proposed. It supports designers in modelling web information systems based on specific types of framework (i.e., Model-View-Controller frameworks, Object-Relational Mapping frameworks, and Dependency Injection frameworks) providing four extended UML Class Diagrams (i.e., Domain Model, Persistence Model, Navigation Model and Application Model).

In the area of static and structural analysis of software programs, including procedural and Object-Oriented paradigms; many graph based abstractions have been formulated.

A *Control Flow Graph* (CFG) [1] is a classical model which provides a structural perspective about the computational flow of a software program, supporting structural testing techniques (e.g., Control Flow Testing). A CFG is a directed graph whose nodes are considered as the basic blocks of a software program (i.e., a linear sequence of instructions executed as an atomic operation), and whose edges represent control flow paths (i.e., conditional jumps or accesses to locations associated with a label).

A *Data Flow Graph* (DFG), also known as Definition Use Graph [90], is an abstraction for supporting structural Data Flow Testing techniques through the identification of *definitions* and *uses* so as to exploit variables occurrences within programs, statically analysing their values and produced side effects. A DFG is a directed graph, interpretable as an annotated CFG, whose nodes can be variable definitions (i.e., *defs*) as in the case of assignment statements, or variable uses (i.e., *uses*), classifiable as read operations of a variable within a predicate (i.e., *p-use*), as in the case of conditional guards, or within computations (i.e., *c-use*), as in the case of a variable used

in external assignments. A DFG edge represents a sequential execution and it can be decorated with the expected side effect for a *c-use* or with the conditional branch for a *p-use*.

These first two abstractions, despite being designed to primarily cover the need for static analysis of procedural programs, opened the way to many extensions and integrations focused on the concept of dependencies, and on the Object-Oriented paradigm.

On the one hand, many works address the problem of representing software structural dependencies among program slices (i.e., decompositions in statements faithfully representing the original behaviour with respect to analysed properties). Starting from the classical literature about structural software testing, in [32,79] a *Program Dependence Graph* also known as *Program Dependency Graph* (PDG) is described, as a graphical representation of a program where nodes model regions of code or single statements, while edges model information about either control dependencies (i.e., dependencies among single statements or groups based on predicate evaluations for conditional executions) within a *control dependence subgraph*, automatically derivable by a CFG [46], or data dependencies (i.e., dependencies among statements induced by data variables assignments) within a *data dependence subgraph*, which can be obtained through a data flow analysis stage.

On the other hand, a variant of PDG has been proposed in [70] so as to cope with Object-Oriented paradigm by defining an additional subgraph, named *class hierarchy subgraph*, composed by program classes as nodes, considering that a class defines objects with data variables, and hierarchies among objects as edges, considering class extensions and method signatures. Also the control dependence subgraph and the data dependence subgraph have been improved and merged within an *Object-oriented Program Dependence Graph* (OPDG), addressing methods invocations among objects as well as polymorphic attributes and calls.

Several approaches addressed the enrichment of aforementioned abstractions so as to accomplish modelling requirements relate to inheritance, polymorphism, and dynamic binding mechanisms.

In [108], a reinterpretation of *def* and *use* concepts in a Object-Oriented perspective for the DFG is proposed, including inter-class relationships among distinct program objects, also describing an inter-class *def-use* analysis tech-

nique subject to a partial representation of monitored classes.

Many language specific enhancements of PDG and OPDG abstractions have been proposed for Object-Oriented programs in the area of program slicing approaches [119]: for C++ [65] and for Java™ [60, 117, 123].

The evolution of programming languages and the increase in functional complexity exacerbate the need for further abstractions enabling representations of dynamic and reusable components features also orientated towards distributed systems and web development, which usually is supported by frameworks and libraries realising automated installation of components through implementations of the Inversion of Control principle.

In [120] the *Component Interaction Graph* (CIG) is proposed to enable representation of collaborative relationships and dependencies among software components within component-based architectures. Data dependencies are captured as the effects of an update process on runtime components, considering that an interface invocation represents the triggering point for executing a group of methods or functions, offered by involved components. Specifically, a CIG is a graph abstraction able to provide a structural overview of the interactions of a component-based system by depicting components interfaces as nodes, and dependencies as edges, conceptually identifiable as events (i.e., interface invocations, user actions, and exceptions).

In order to support reliability analysis processes over component-based applications, in [122] a probabilistic model, adapted from the CFG principles for capturing architectural dependencies among components, is proposed. This abstraction, named *Component-Dependency Graph* (CDG), is a directed graph whose nodes are components, decorated with their estimated reliabilities and their average execution times, and whose directed edges are transitions between components, each one in turn annotated with details about its estimated reliability and its execution probability.

In [105] a *Dependency Call Graph* is proposed to represent key aspects for the modernisation towards Service-Oriented Architectures of monolithic legacy systems, developed within the Java™ Enterprise Edition ecosystem, exploiting web servlets, JavaServer™ Pages, and JavaServer™ Faces specifications on 3-tier architectures. A *Dependency Call Graph* depicts page controllers as graph nodes, interconnected by edges figuring navigation transitions (i.e., links) among pages. Specifically, dependencies which commonly remain hidden (due to various factors, such as connections among multiple

tiers, configuration files with *ad hoc* syntax, heterogeneous source code fragments) are expressed through a language-independent meta-model termed *Knowledge Discovery Meta-Model* [83], enlightening dependencies related to containers regulated through Remote Method Invocations.

Above all, these component-based graph abstractions are not sufficient for modelling dependencies within applications exploiting Dependency Injection (DI) and automated contexts management, lacking in expressiveness about components scopes, their visibilities, and their lifecycles boundaries as well as proxy and interceptor entities, automatically acting in background through a DI container.

Finally, the Java™ Enterprise Edition ecosystem provides the so called Contexts and Dependency Injection (CDI) specification [91,92], which comes with the built-in concept of *bean dependency graph* (abbr, *bean graph*). This abstraction is a directed graph showing dependency relationships among distinct components (i.e., beans contextual instances) managed in background by the CDI container. The graph has been designed, specifically, for enlightening fine-grained characteristics of CDI components and for depicting injection points, types, scopes as well as proxies, interceptors, qualifiers, and producer methods (graph vertices may be *contextual instances*, type declarations of injection points or effective injected types at runtime, while edges represent dependency relationships). The graph can be built dynamically at runtime, starting from source code and exploiting a reflection mechanism offered by CDI, enabling software components introspection.

For the research presented in this thesis, the information provided by the *bean graph* was a source of inspiration, considering the salient characteristics of managed components within Web Application controlled by a DI container.

3.1.2 Navigational and behavioural design

Many models and abstractions have been proposed in literature, also, for capturing behavioural characteristics of software programs, representing dependencies and relationships among runtime instances of classes or components, with the aim of describing the expected behaviour from a functional perspective.

The rise of Web Applications, subject to different Enterprise architectural styles (e.g., monolithic, service-oriented, microservice-oriented), regulated by Internet protocols, and deployed on remote Application Servers

(where backend and frontend modules may operate independently or as a whole) exacerbates the need for modelling their navigational characteristics. Consequently some semi-formal and formal standards have been introduced; in this subsection, the most significant ones for the research of the thesis are mentioned.

In the practical experience, functional aspects of Web Applications are expressed through a simple and intuitive abstraction, named *Page Navigation Diagram* (PND) [63], characterising navigation design salient features through the definition of a finite state machine where web pages act as states, while hyperlinks act as transitions.

Usually, in component-based Web Applications for each page exists a controller (i.e., a component responsible of handling main interactions) which maintains the state. In so doing, a PND can drive incremental design approaches defining controllers implementations from a navigational perspective or, vice versa, they can be automatically extracted starting from more detailed artefacts (e.g., Object Relation Diagrams or UML Robustness Diagrams).

In [63], an approach for constructing a PND from an *Object Relation Diagram* (ORD) [62] is presented; the ORD, originally introduced for modelling Object-Oriented programs, is a directed graph where nodes represent instantiated objects (e.g., the web page controllers), while edges represent relationships among objects (e.g., the links among web pages).

In many design and development methodologies, functional aspects of software programs are expressed through more useful and extensible diagrams abstractions, frequently relying, also in this case, on UML [11] standard, which offers a wide range of useful and extensible diagrams, enlightening key behavioural aspects within different perspectives. Component-based applications privilege diagrams with a dynamic nature, such as: *Activity Diagrams*, for modelling control flows among object instances; *Statechart Diagrams*, for modelling state machines; *Use Case Diagrams*, for modelling application use case scenarios, defining actors and functional aspects within the operative context; *State Machine Diagrams*, for modelling different states of an object instance during execution; *Collaboration and Sequence Diagrams*, for modelling objects interactions driven by method invocations, and *Robustness Diagrams*, widely adopted within ICONIX-based Software Engineering devel-

opment processes, for supporting designers in modelling interactions among actors, pages and components complying with application use cases [98].

In particular, leading the robustness analysis, the Robustness Diagram aims at discovering and identifying involved actors among use cases, bridging the gap from analysis to design in order to define domain model, business logic and pages reachability through its main elements: *entities*, representing domain model objects; *boundaries*, representing web pages; and, *controllers*, implementing the business logic of the application by representing invocable page methods.

Each element of the diagram must be interconnected following four connection rules: *i)* an *actor* interacts directly only with *boundaries*, *ii)* a *boundary* interacts only with *controllers* or *actors*, *iii)* a *controller* can interact with any other element, except for actors, and *iv)* an *entity* can interact only with *controllers*, which manipulate the domain model. The Robustness Diagram subtends a reachability graph, decorated with dependency relationships, highlighting interactions among end-users and page controllers and figuring all designed navigation rules for each modelled use case, thus mixing information derived both from functional and structural perspectives.

The increasing interest in Web Applications development have led the researchers in providing useful abstractions for modelling, in a static or dynamic perspective, internal behaviour of this family of softwares, also adopting structural models (e.g., UML Class Diagrams) with the aim of capturing both structure and navigation data flows.

In [94], a meta-model for describing a Web Application is provided through UML Class and Object Diagrams, identifying salient information of a web site (i.e., web pages, hyperlinks, input forms, frames) distinguishing between static or dynamic pages (i.e., considering that page content may depend on end-users inputs, thus determining the presence of navigation conditional rules). The proposed meta-model opens the way to a static analysis stage of a web site, so as to understand its organisation (in terms of navigation paths and allocated page variables), and to a dynamic validation stage, so as to execute white box testing with *ad hoc* coverage criteria inspired by Data Flow Testing (i.e., page testing, hyperlink testing, definition-use testing, all-uses testing, and all-paths testing). In this work, the abstraction adopted within test case selection is a graph abstraction derived from the UML Object Diagram instances of the web site, whose nodes correspond to objects

(e.g., pages or forms) and whose edges represent links between pages. This work does not consider Web Applications built over a DI container or with automated contexts management capabilities, which are instead within the scope of this thesis.

In [84], a framework for supporting developers in the design of Web Applications is proposed; it focuses on user experience (e.g., usability), through the definition of a UML extension, fitting stakeholders' goals and adopting a user-oriented semantics. Specifically, UML Class Diagrams have been enriched to model structural and navigational aspects of web pages through the adoption of custom stereotypes (e.g., screen template, layout content, link) under different perspectives (e.g., isolated views or the overall application).

3.2 Model-Based Testing

In many disciplines and software development methodologies, testing practices have become a standard within enterprise organisations to verify the correctness of a System Under Test (SUT) with respect to expected behaviours. In Information Technology and Computing areas, a system can be often considered as an application program based on a software implementation, thus is common the case of naming a SUT also as Implementation Under Test (IUT).

Among the plethora of testing approaches, Model-Based Testing (MBT) [3, 28, 114] is a widely adopted technique, exploiting formal and semi-formal models as primary documentation artefacts leading the choice of a stimulus (or a sequence) to the SUT and its verification, also in conformance with coverage criteria describing the confidence level in the absence of defects. MBT uses models to describe the behaviour of a system and it can be considered as a specialisation of Model-Driven Engineering (MDE) [101], producing benefits in contribution to the quality of functional requirements, to the (automated) generation of tests and systematic coverage of test suites. [66]

In so doing, MBT may demand for the collaboration of different technical experts for describing different aspects of the same SUT in different perspectives with different level of granularities.

The research addressed in this dissertation adopts the following taxonomy, inspired by the works in [5, 88]:

- an **error** is a runtime deviation of the system state from the expected

one, bringing the system into an erroneous state which may, or may not, disrupt the delivered service;

- a **failure** occurs when a delivered service deviates from the expected behaviour; thus a failure can be considered as a manifestation of an error of the SUT;

- a **fault** is the (internal or external) cause of an error. A fault at source code level is called *defect*. In turn, a defect at implementation level is called *bug* while at design level it is called *flaw*.

In this Section, a description and an evaluation of how presented structural and behavioural abstractions are adopted as fundamental models within main structural and functional testing approaches is reported.

3.2.1 Structural Testing

In a *white box* perspective, structural testing techniques verify correctness of Implementations Under Test by exercising and comparing their behaviour with respect to their concrete implementations and their expected behaviours, exploiting the source code or some modelling abstraction for test case generation and selection (e.g., Control Flow Graph, Data Flow Graph). Note that structural testing techniques, basing their foundations on how a software program is effectively implemented, may suffer from a tautology problem: when an implementation is defective, its defects may affect, in turn, the test case selection so as to be ineffective in finding defects; in this case, tests may not discover defects.

Among structural testing techniques based on Control and Data Flow Graph abstractions, the most relevant in literature are *Control Flow Testing* (CFT) and *Data Flow Testing* (DFT).

CFT techniques [9] aim at covering different paths of control flow, thus exploiting a CFG abstraction, relying on some coverage criteria (e.g., *All Nodes*, *All Edges*, *All Conditions*, *All Paths*). In so doing, CFT approaches enable the identification of a coverage analysis measure representing an estimate in the absence of residual defects on the identified complexity under the chosen coverage criterion.

The DFT methodology, proposed by Rapps and Weyuker [90] and later enhanced in [39], extends the CFT approach by exercising data dependencies

among variables, exploiting the DFG abstraction and defining new coverage criteria (e.g., *All Defs*, *All Uses*, *All DU-Paths*, *All Paths*) within a single procedure of a program. A first approach overcoming limitation of intra-procedural DFT has been proposed by [48] as an inter-procedural DFT approach to capture dependencies derived by function invocations, thus implemented within distinct procedures. Later, various solutions [27,47,107] have been proposed so as to adapt DFT for the case of Object-Oriented programming, thus covering *def-use* couples at different levels of granularity by modelling also relationships among attributes and methods of different classes. In [69], the approach is further extended to the case of web components covering couplings occurring in web interactions due to values exchanged in HTTP requests/responses, in XML documents, or stored within HTML documents. In [67, 124] structural approaches based on Control and Data Flow Testing focusing on aspect-oriented programs (specifically implemented for AspectJ) have been presented, respectively, for unit or integration testing.

In [121], a methodology exploiting the CIG abstraction for testing component-based applications, focusing on detecting integration faults which may be activated by interactions among components, is presented. The methodology provides a test case selection strategy for integration testing, assuming that each component has been already tested through a unit testing stage. It relies on a fault model which classifies faults in three typologies: *i)* inter-components faults (i.e., faults resulting from combined uses of distinct components, indistinguishable when they operate separately), *ii)* interoperability faults (i.e., faults resulting from interactions among components built under different infrastructures, operating systems, programming languages or specifications), and *iii)* traditional faults (i.e., faults which can be isolated within a single component). In background, a CIG is adopted and derived through a static analysis about components interactions, evaluating events and data flow exchanged among components interfaces (e.g., method invocations).

3.2.2 Functional Testing

In a *black box* perspective, functional testing techniques verify the conformance between an Implementation Under Test and a specification, neglecting structural aspects of a system (e.g., the source code) in favour of the adoption of functional abstractions such as software requirements or use cases, describ-

ing application business scenarios. In this context, navigational design abstractions become even more relevant for testing component-based Web Applications, exploiting use cases information so as to model the expected application behaviour beyond unconstrained end-user interaction flows, which are driven by hyperlinks. In so doing, use cases represents the major source of functional information. [113]
Among functional testing practices, a significant notation category is the *scenario-oriented*, also known as interaction-oriented, which describes from the end-user perspective all reasonable runtime interactions between the IUT and the sequences of inputs or outputs.

In [24], *Message Sequence Chart* abstraction, whose information is quite adaptable to *UML Sequence Diagrams*, is adopted to define a conformance testing technique generating an *ad hoc* test suite. In [116], *UML Activity Diagrams*, each one related to a single use case, are annotated with custom test data requirements enabling the definition of a GUI testing approach which relies also on custom coverage criteria (e.g., happy path, round trip). In [76], the authors present a relevant work describing an approach automating the generation of test scenarios for Object-Oriented systems in embedded environments, starting from formal requirements specifications and a custom extension of *UML Use Case Diagrams* and templates, thus proposing a *requirement-by-contract* approach. In [55], the proposed test generation approach adopts *UML Use Case Diagrams* in conjunction with *UML Class Diagrams*, decorated with guards, invariants and post-conditions, as input specifications in order to generate verification sequences over mutated *UML Object Diagrams* within the IUT. In [56], *UML Use Cases*, more specifically textual use case templates, are the main abstraction proposed to apply a Model-Based Testing methodology, leveraging a domain-specific modelling language, empowered by special low-level keywords referred to User Interface elements (e.g., click button X) and high-level action words (e.g., take picture), subtending the sequence of actions to be performed within a use case. In [110], the automated generation of a suite of integration test cases is performed through the combination of *UML Collaboration Diagrams*, logic contracts capturing expected post-conditions, and an additional artefact named *execution tree of components*. This enables an *end-to-end* testing approach covering the activities along the entire testing process, but it does not address dependencies managed by Dependency Injection containers or, more in general, generated realising the Inversion of Control principle.

Another significant notation category is the *state-oriented*, which describes the IUT by reactions on inputs and outputs through finite state automata abstractions, laying its foundations on the consideration that the behaviour of a system can be fully abstracted by its state (i.e., the automaton current state) and the invoked operation (i.e., the selected output of the current state).

State-oriented testing techniques cannot be *a priori* classified as purely functional testing, considering that its classification depends on how states are derived and what they effectively model. In the frequent case where states are derived through a functional analysis (e.g., from use case templates in pre-conditions, post-conditions and behavioural descriptions), state-oriented testing techniques can be considered as a special case of functional testing. [49]

In [12, 78] a technique for generating test cases from *UML Statecharts* are presented; in both works test data have been generated adopting two different tools (i.e., respectively Rational Rose and Leirios Test Designer). While the first work adopts *UML Statecharts* in isolation, the second accompanies them with *UML Class and Objects Diagrams*.

In [93] a Model-based Testing technique to test web frontends (i.e., to be indented as standard web sites written in HTML without considering back-end modules) is presented. A major contribution of the paper is the proposal of a syntax leading the specification of *UML Statecharts* for Web Applications, resulting in a *grey box* perspective adopting a structural approach over behavioural and navigational artefacts.

In [13,61], a functional test approach implemented as a tool (i.e., UniTesK) to automatically derive test sequences by analysing paths over a finite input/output state automaton, deduced by program contracts, is described. In so doing, the approach enables automated generation of test scenarios on the basis of relevant system operations, described in terms of pre-conditions, post-conditions, parameter types, and invariants.

In [86], automated test case selection for RESTful web services is performed in a model-based approach exploiting manually user defined *UML State Machine* specifications of the expected behaviour, further decorated with state invariants and state post-conditions.

3.3 Non-Model-Based approaches

Several approaches and techniques have been proposed in literature for supporting test case generation [3] exploiting not only models but also different sources of information as input artefacts (e.g., the program structure, the source code, the information about input/output data space, the dynamic data generated during execution).

In this Section, for the sake of completeness, a brief overview of main non-model-based test case generation approaches is reported.

A first category is represented by *symbolic execution* approaches [16, 42, 59], directly exploiting source code analysis in order to automatically generate test data, in a purely *white box* perspective. Program variables are represented as symbolic expressions whose inputs are symbolic values, instead of concrete ones. During the execution, a state of the program under test is maintained; the state includes the symbolic values of the instantiated program variables, a so called path constraint (i.e., a Boolean formula which has to be satisfied for executing the program over symbolic inputs exercised during a path), and a program counter (i.e., a pointer to the next statement of the program to be executed). All the available and enumerated states contribute in the definition of a *symbolic execution tree*, representing the hierarchy of *execution paths* encountered during the effective execution of the program. Symbolic techniques applied over large-size programs may suffer mainly of problems about path explosion, path divergence or complex constraints.

A second category is represented by *combinatorial testing* techniques [14, 20, 23, 30] which exploit heuristics to approximate parameters and inputs modelling them as sets of factors and values, thus covering a subset of combinations of the elements characterising the SUT. In so doing, a software program can be tested selecting a sample of possible input parameters (i.e., a specific subset of its available configurations, also called instances) which have been combined together, also considering the available fields within the User Interface. One of the most popular implementation of this category is the so called *combinatorial interaction testing* [85], introducing the concept of *covering arrays* for modelling the combinations of settings of a program, where each row can be considered as a specific test case, thus supporting the sampling stage. Mathematical and statistical research areas have con-

tributed in the definition of methods and algorithms for generating arrays of samples, with suitable program features.

A third category is represented by *search-based software testing* techniques [45, 118], exploiting optimisation algorithms which automate the generation and search for test data and inputs to final test cases. These approaches rely on the definition of fitness functions, modelling the test objectives fixed for the SUT and constituting a basis for the implementation of *ad hoc* search algorithms, both in structural or functional perspectives, aiming at maximise the goals and, simultaneously, minimising the costs (e.g., the oracle cost). [74]

A fourth category is populated by techniques based on *random testing* [44]; one of the most popular testing method which performs randomly the choice of input test data, in a merely *black box* perspective. In such approaches, the generation of independent inputs is usually delegated to a random or pseudo-random generator whose output results - for each test case - are then compared with the designed program specifications. Within this category, *adaptive random testing* techniques [18] have been proposed as an enhancement, aiming at distributing test cases more evenly within the input domain space through the definition of *ad hoc* metrics.

Finally, two categories inspired by random testing theory have emerged.

On the one hand, *mutation testing* [50, 80] is a fault-based technique, operating in a *white box* testing perspective, which leverages on the concepts of mutation (i.e., a small syntactic change on the source code of the SUT), mutant (i.e., a faulty version of the SUT affected by a mutation), and mutation operator (i.e., a transformation rule which generates a mutant, modifying variables and expressions by insertion, replacement, or deletion). The primary intent of this technique is the evaluation of the effectiveness of a test suite, in terms of fault detection capabilities, defining a mutation adequacy score. In general when a same test suite is executed against a mutant, it is possible to understand its robustness; a robust test suite should catch injected faults within mutant version, by having at least one failing test case. Mutation testing can be applied at different testing levels (i.e., unit, integration or specification).

On the other hand, another fault-based technique is *metamorphic testing* [17,104], proposed as an approach for test case generation which exploits

the input-output pairs of previous successful (or not) test cases with their related types of errors for generating new test cases. In so doing, such techniques base their test case generation strategies on the assumption that a kind of evolution among test cases is the foundation for discover undetected errors (i.e., all the possibly errors which have not been detected in previous successful test cases), exploiting existing metamorphic relations (i.e., relationships generated among multiple executions of the SUT). This technique may also help test result verification stages, alleviating the oracle problem [6], and it can be applied in conjunction with test case selection strategies in *black box* or *white box* perspectives [125].

Chapter 4

Running Case Study

For the sake of clarity and to support the reader in understanding the problem and the examples presented within the dissertation, in this Chapter, a prototype stateful Web Application, code-named "Flight Manager", is described.
Technical and design choices, leading the implementation of this application will be also useful in defining salient characteristics of Dependency Injection and automated contexts management mechanisms, exemplifying the proposed fault model (Sect. 5.3) and the proposed methodology (Sect. 6.2) applying it to a concrete state of the art application.

In particular, the operative context is described in Sect. 4.1, the functional design in Sect. 4.2, the architecture in Sect. 4.3, and the navigation design in Sect. 4.4.

4.1 Operative context

Flight Manager[1] is a Web Application accounting the operative context of an online flight booking IT system, available through the Internet.

The platform has been designed to cover functional requirements of three user classes:

- not registered users (i.e., visitors), exploiting offered services as "one-time" accounts without the need for authentication;
- users with a premium account (i.e., registered), consuming exposed services, only, after a login authentication process;
- administrators (i.e., admins), accessing a reserved area dedicated to managing entities related to "for sale" products (i.e., flight tickets).

The application domain model, represented in Fig. 4.1 through an *UML Class Diagram*, captures from a conceptual perspective the existing relationships among the fundamental entities (i.e., *User*, *Booking*, and *Flight*).

A booked ticket (i.e., *Booking*) is characterised by its issuing date, its price (i.e., by list or after a discount), an internal identifier (i.e., a secret useful to unregistered users for retrieving and managing booked tickets at any time, before the flight), a list of passengers (i.e., considering also the case of a single applicant which buys more tickets in a single booking transaction), and related outbound (and return) flights information.

An available and scheduled *Flight* represents a single travel from a source *Airport* to a destination one, and it is characterised by an internal code, the nominal timetable (i.e., the dates and times related to departure and arrival), the availability in terms of seats (i.e., distinguishing between total capacity and actual reserved quantity), and the suggested price per passenger.

Optionally, each booking can be associated with a buyer account, offering extra privileges to registered users in terms of future promotions and discounts based on historical purchasing data. A *User* account is characterised by some credentials (i.e., username and password) and a role (i.e., *UserRole*), distinguishing between premium customers (i.e., *REGISTERED*) or administrators (i.e., *ADMIN*).

[1] Publicly available at `https://github.com/STLAB-DINFO/flight-manager`

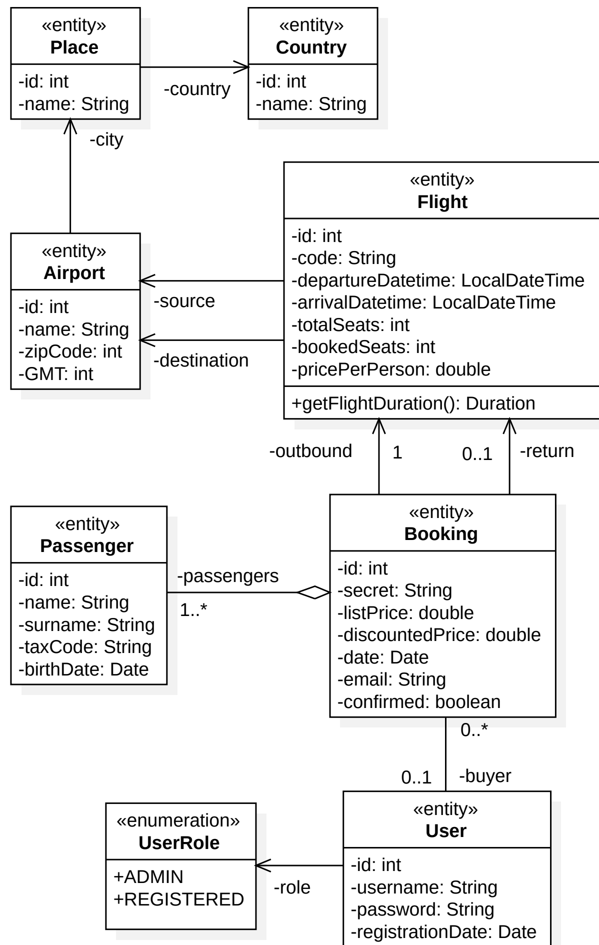

Figure 4.1: UML Class Diagram of *Flight Manager* domain model.

4.2 Functional design

From a functional perspective, the use cases of *Flight Manager* have been classified by user role.

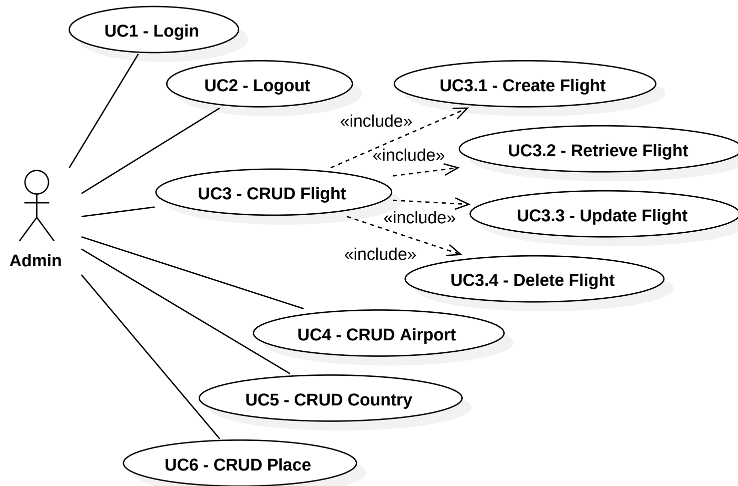

Figure 4.2: UML Use Cases diagram of *Flight Manager* administrator users.

On the one hand, see Fig. 4.2, an administrator is authorised to manage through *ad hoc* CRUD (i.e., Create, Retrieve, Update, and Delete) operations all the available system entities (i.e., *Flight*, *Airport*, *Country*, and *Place*).

On the other hand, see Fig. 4.3, the main use case for a customer (visitor or registered) consists in booking a flight (i.e., UC8). This action, practically, consists in searching a flight (i.e., UC7), selecting a starting and an arrival airport, a departure date (and possibly a return one) and finally, declaring the number of desired tickets. Once the system has returned the query results, the user chooses a specific flight (i.e., UC8.1) and, after entering the required data (i.e., UC8.2), the system presents the summary of the ongoing reservation, in order to complete the payment (i.e., UC8.3). In case of user confirmation, the system requires the specification of an email address which can be used in combination with the secret booking code to later access a reserved area, where to consult passengers and flight data (i.e., UC9), print tickets (i.e., UC9.1) or activate a cancelling procedure (i.e., UC10).

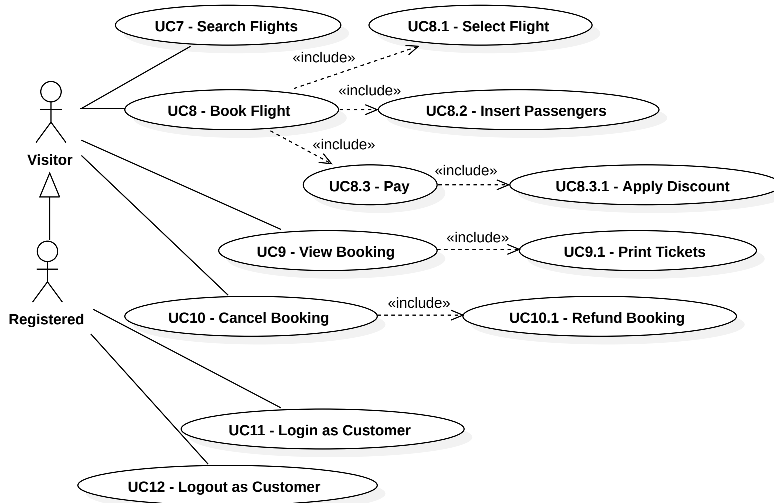

Figure 4.3: UML Use Cases diagram of *Flight Manager* customers users.

Note that, from the use cases perspective, the only difference between a visitor and a registered user consists in the ability of performing the authentication process (and of course the logout process as well); but in practice, the application will consider in different manners their navigation experiences within the platform (i.e., some algorithms will change their behaviour with respect to historical data stored for premium accounts).

4.3 Architecture

From an architectural perspective, *Flight Manager* has been designed as a classical 3-tier *stateful* architecture developed through the Java™ Enterprise Edition (JEE) ecosystem, with the following specifications:

- *JavaServer™ Faces* (JSF), defined within JSR-314 [15], for the presentation layer, building server-side User Interfaces (i.e., view pages) populated with data provided by running *stateful* components (i.e., page controllers). JSF is the standard component-oriented UI framework for JEE.
  The underlying implementation of JSF is *Oracle Mojarra*;

- *Contexts and Dependency Injection* (CDI), originally defined within JSR-299 [58], for the business logic layer, with the responsibility of DI framework, enabling type-safe resolution of managed components (e.g., page controllers or collaborative ones) and automated injection mechanisms, also binding their lifecycles to available built-in contexts (i.e., *@RequestScoped*, *@ConversationScoped*, *@SessionScoped*, *@ApplicationScoped*, and *@Dependent*).
  The underlying implementation of CDI is *JBoss Weld*;
- *Java™ Persistence API* (JPA), defined within JSR-317 [26], for the data-access layer, managing persistence and Object Relational Mapping (ORM) processes, bridging the gap between an Object-Oriented model and a relational database schema.
  The underlying implementation of JPA is *JBoss Hibernate*.

As can be observed in Fig. 4.4, the presentation layer is strongly coupled with backend modules provided by the business logic layer, which in turn is populated by a collection of page controllers (e.g., *LoginController*, *AirportController*, *RegisteredBookingController*) and other task-specific components (e.g., *BillingComponent*, *TemporaryReservationComponent*, *DiscounterComponent*).

As standard in *stateful* applications, these managed components implement specific use cases and depend on the data-access layer in order to interface database records and mapping them in the form of domain model entities. For these purposes, an intermediary role is played by a group of collaborative components (e.g., *AirportDao*, *BookingDao*, *FlightDao*, *PassengerDao*), named Data Access Objects (DAOs), providing an abstract interface towards the underlying Database Management System (DBMS), also exposing a set of methods to perform CRUD operations on the relational tables. To accomplish these tasks, each DAO depends on a special JPA component, named *EntityManager*, which allows querying of entities within database transactions.

Three separate considerations must be spent for the following task-oriented managed components: *i) BillingComponent* has the responsibility of determining the fee of each emitted flight ticket, applying a variable tax value considering the country of arrival; *ii) DiscounterComponent* has the responsibility of applying dynamic discount strategies over listing prices of flights, exploiting, in turn, external task-specific components which extend

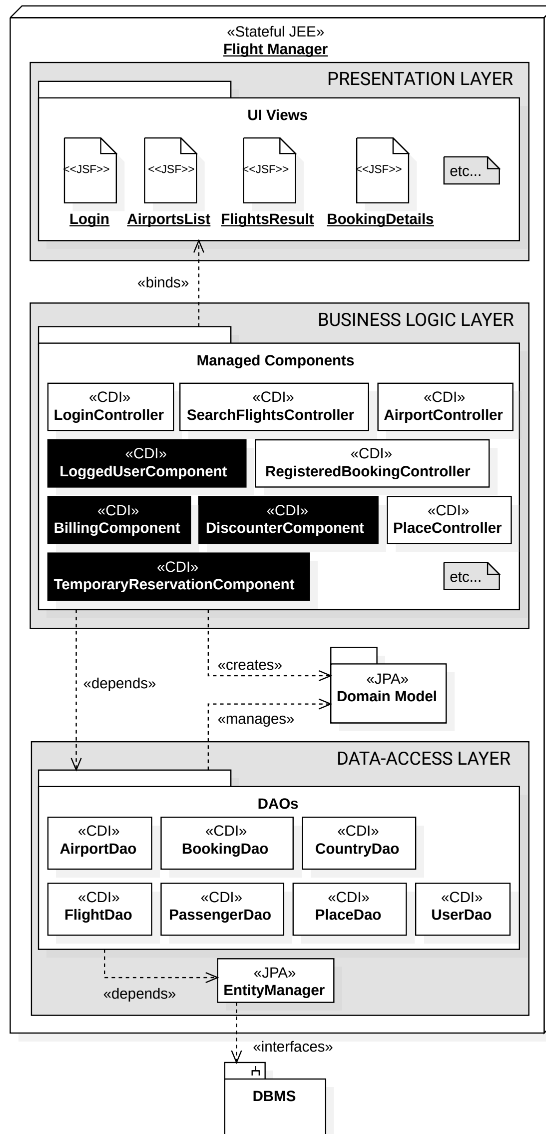

Figure 4.4: 3-tier architecture overview of *Flight Manager*, as *stateful* application with a set of managed components responsible of business logic (i.e., page controllers and task-specific components) and a set of data-access components (i.e., Data Access Objects).

a superclass named *DiscountStrategyComponent*[2] (i.e., *BaseUserDiscount*, *BigGroupDiscount*, *CrazyWednesdayDiscount*, *GoldUserDiscount*, and *SilverUserDiscount*); *iii) TemporaryReservationComponent* has the responsibility of keeping track of the ongoing booking processes, not yet confirmed, so as to reserve the seats until the user completes the use case (in so doing, seats cannot be stolen by other users). This component cooperates with another task-specific component, named *TemporaryReservationRepository* which lives within the *application* context and has the responsibility of maintaining an *in-memory* database of current total reservations of *Flight Manager*.

In Tab. 4.1, the list of managed components, designed and implemented within *Flight Manager*, is reported; for each component, the category (i.e., data-access, page controller, or task-specific) and the belonging context (distinguishing between the adopted CDI annotation and the interpreted context) are documented.

[2] Each discount strategy, extending *DiscountStrategyComponent*, applies a custom policy; the *BigGroupDiscount* takes into account the number of passengers within a booking and decides if they represent a "big group", the *CrazyWednesdayDiscount* applies a discount if and only if the day of the week is Wednesday, while *BaseUserDiscount*, *GoldUserDiscount*, and *SilverUserDiscount* consider the purchasing history of a logged user, thus rewarding the affiliation level.

Component	**Category**	**CDI scope**	**Context**
AirportController	page controller	*@ConversationScoped*	*use case*
AirportDao	data-access	*@RequestScoped*	*request*
BaseUserDiscount	task-specific	*@RequestScoped*	*request*
BigGroupDiscount	task-specific	*@RequestScoped*	*request*
BillingComponent	task-specific	*@SessionScoped*	*session*
BookingDao	data-access	*@RequestScoped*	*request*
BookingLoginController	page controller	*@RequestScoped*	*request*
BookingSessionComponent	task-specific	*@SessionScoped*	*session*
CountryController	page controller	*@ConversationScoped*	*use case*
CountryDao	data-access	*@RequestScoped*	*request*
CrazyWednesdayDiscount	task-specific	*@RequestScoped*	*request*
DiscounterComponent	task-specific	*@RequestScoped*	*request*
FlightController	page controller	*@ConversationScoped*	*use case*
FlightDao	data-access	*@RequestScoped*	*request*
FlightManagerComponent	task-specific	*@ConversationScoped*	*use case*
GoldUserDiscount	task-specific	*@RequestScoped*	*request*
LoginController	page controller	*@RequestScoped*	*request*
LoggedUserComponent	task-specific	*@SessionScoped*	*session*
PassengerDao	data-access	*@RequestScoped*	*request*
PasswordManagerComponent	task-specific	*@RequestScoped*	*request*
PlaceController	page controller	*@ConversationScoped*	*use case*
PlaceDao	data-access	*@RequestScoped*	*request*
RegisteredBookingController	page controller	*@SessionScoped*	*session*
RouterComponent	task-specific	*@ApplicationScoped*	*application*
SearchFlightsController	page controller	*@SessionScoped*	*session*
SilverUserDiscount	task-specific	*@RequestScoped*	*request*
TemporaryReservationComponent	task-specific	*@Dependent*	*prototype*
TemporaryReservationRepository	task-specific	*@ApplicationScoped*	*application*
UserDao	data-access	*@RequestScoped*	*request*
VisitorBookingController	page controller	*@ConversationScoped*	*use case*

Table 4.1: Overview of *Flight Manager* managed components, each one with the indication of its category (i.e., data-access, page controller or task-specific), its designed CDI scope and the related context.

4.4 Navigation design

Many disciplined software development practices for Web Applications adopt (semi-)formal design artefacts (e.g., Page Navigation Diagrams, UML Robustness Diagrams) describing end-users interactions and transitions among views (i.e., web pages), thus leading the early design of use cases in a navigational perspective which outlines expected and feasible navigation paths.

These abstractions include oriented edges, labelled with the indication of actions leading the navigation, and more than one label may be present on the same edge; in these cases, it means that two different operations, handled differently by the starting page controller, forward to the same landing page.

The Page Navigation Diagram of *Flight Manager* is reported in Fig. 4.5.

Specifically, each end-user interfaces the application starting from the *Home* page, which exposes a set of useful hyperlinks. An unauthenticated user performs the authentication process from the *Login* page: on the one hand, admin users can exploit administration functionalities, depicted in the upper-part of the diagram, starting from the *AdminPanel* page; on the other hand, registered customers can browse the pages in the lower-part of the diagram, except from the *BookingLogin* page, which is dedicated to visitor accounts using the secret booking code to access the reserved area for managing their bookings.

The main use cases for a visitor or registered user are implemented within these six pages: *Login*, *Home*, *FlightsResult*, *FlightDetails*, *BookingDetails*, and *Confirmation*.

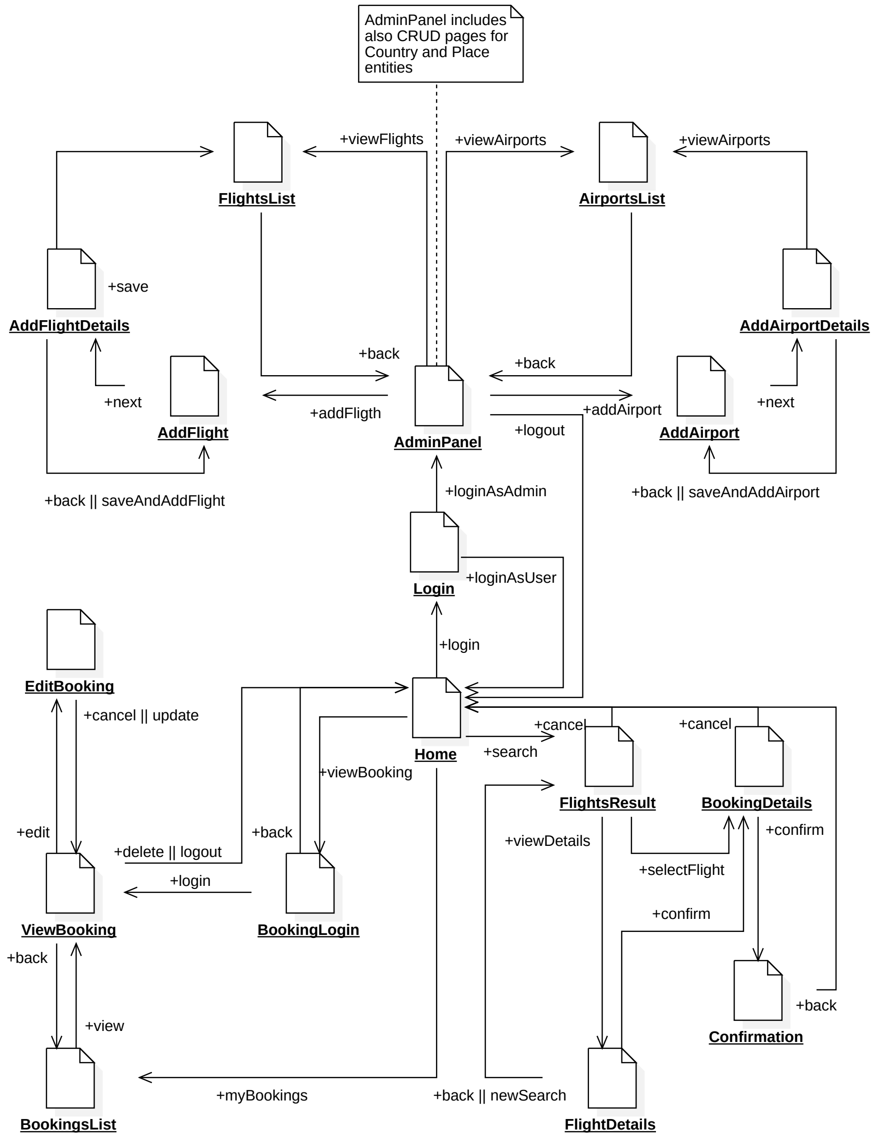

Figure 4.5: Page Navigation Diagram of *Flight Manager*.

Chapter 5

Motivations and Problem Formulation

In this Chapter foundations, leading the research in the definition of an effective methodology for the verification of Web Applications designed and developed within Enterprise Architectures with a stateful behaviour exploiting Dependency Injection and automated contexts management mechanisms, are provided.

Specifically, a detailed description of motivations with respect to common characteristics of stateful architectures and DI frameworks is reported in Sect. 5.1, also enlightening salient considerations about the HTTP protocol operating in the client-server paradigm.
The problem formulation is reported Sect. 5.2, considering components couplings across contexts, as well as underlying basic assumptions.
Finally, the fault model at the core of the proposed methodology is reported in Sect. 5.3, providing both a conceptualisation of identified faults and their concretisation within the running case study of Chapter 4.

5.1 Motivations

In the design and development of modular software architectures, dependencies management of software components becomes a key aspect of the overall application complexity. To mitigate it in a productive way, dependencies management is often delegated to a *container*, assuming the responsibility of taking care of components creation and dependencies installation by injecting required components in dependent ones, in compliance with their declared types.

This practice, commonly known as Dependency Injection (DI), is supported by many programming languages and provided in widely adopted implementations and high-level DI frameworks with additional features and responsibilities in facilitating components lifecycle management.

In particular, the container may bind components lifecycle to predefined *contexts* (e.g., *request*, *session*, *application*), defining visibility boundaries, handling construction and destruction policies, as well as controlling the so called *injection points* (i.e., lines of code where dependencies have been defined and the injection of component instances occurs within dependent components, such as on their class attributes or on their method parameters). For the sake of conciseness, this dissertation will refer to these responsibilities as *automated contexts management.*

While mediating and facilitating interactions among components, the automated contexts management produces *shared dependencies* scenarios, where several dependent components, living within different contexts, may depend on the same contextual instances of injected components, thus sharing their runtime states.

These kinds of couplings among distinct components should be evaluated and considered for an exhaustive testing stage through the definition of *ad hoc* strategies; indeed, on the one hand, standard unit testing does not represent an effective solution in so that mocking dependencies inhibits the manifestation of coupled behaviours and, on the other hand, it may not be feasible to test all combinations of couplings considering dependency chaining scenarios which may be generated according to ongoing end-users interactions along available use cases, involving also more than two components (also indirectly interconnected). Testing strategies specifically characterised on Web Applications paradigms are required for reducing test efforts.

Indeed, DI combined with automated contexts management is further emphasised in Web Applications designed as *stateful* architectures, where interactions among end-users and software components are modelled through client-server paradigm, usually laying on HTTP protocol, maintaining client information in the server-side among multiple HTTP requests and realising use cases within navigable web pages, interconnected via hyperlinks.

This behaviour, in contrast with *stateless* architectures (e.g., REST), allows to store client information within dedicated server-side contexts, improving network performance and decreasing repetitive data sent, exchanged within a series of distinct HTTP requests. This may produce many advantages in applications with a relevant computational effort or in applications subject to strict security constraints.

In such a perspective, the *state* of an application is represented by alive *in-memory* components; in particular, business logic is implemented through specific managed components, named controllers, which are exploited by the presentation layer (i.e., the view pages) to respond to events (e.g., a data input stream) and user interactions (e.g., mouse click action) enabling page transitions, constrained to designed page navigation rules, or internal state modifications.

From the end-user point of view, the application state can be summarised, in reference to each element of the *Model-View-Controller* [25] architectural pattern, as:

- the runtime object instances of the domain logic allocated server-side (i.e., Model);
- the actual visited HTML page (i.e., View);
- the instantiated dependencies hierarchy of the managed component which controls in background the page (i.e., Controller).

Note that, according to the *Page Controller* pattern [35], each logical page is handled by a single managed component of the Web Application. Thus, a page controller subject to automated contexts management is effectively a software object instance, constrained to a specific context and characterised by several dependencies, which in turn may live in different contexts and depend on other components, giving raise to a complex and recursive process of objects creation and referencing, in many cases intermediated by *proxy* objects.

Placing the application state on the server-side, increases the server control over the application behaviour; but, at the same time, the ability to control and detect cases of wrong lifecycle management, unexpected user interactions, or hidden dependencies is reduced, raising also the presence of potential implementation defects, which could lead to application faults (e.g., memory leakages, data inconsistencies, and null pointer exceptions).

Inter alia, automated contexts management, usually, can be configured through meta-information beyond the semantics of the adopted programming language and DI framework (e.g., annotations decorators, as in the case of the List. 5.1 for the JEE ecosystem exploiting the CDI framework) identifying injection points and inferring components lifecycle and visibility within the source code. In this way, although the DI container may solve, in background, dependencies at application initialisation time (e.g., compile-time, build-time, or run-time), automatically binding components instances to predefined contexts relieving developers from this burden (e.g., avoiding a "glue code" style), the overall subtended implementation complexity grows.

The decoupling, realised by the DI container, between dependencies and theirs configurations within source code implementation (i.e., between the type specified in the injection point and the concrete object type instantiated by the container, also exploiting polymorphism provided by many programming languages) slightly blurs the developer's overview about the concrete overall structure of the application, thus exacerbating the complexity in defining an effective test suite.

Finally, List. 5.1 practically highlights how configurations are scattered in the application source code by reporting an example of an annotated CDI component, named *LoginController*, related to UC11 of the case study presented in Sect. 4, defined so as to live within a *request* context, injecting three managed components as dependencies (i.e., *LoggedUserComponent*, *UserDao*, and *PasswordManagerComponent*), starting a new user *session* in the case of successful authentication for registered users.

Note that belonging contexts and injections are defined through *ad hoc* annotations (i.e., respectively *@RequestScoped* and *@Inject*) and the developer does not keep control of dependency characteristics of external components (i.e., the developer can only see that runtime contextual instances will be compliant with specified types without knowing belonging contexts,

concrete implementations or eventual sharing scenarios), while operating on the source code of the dependent component (i.e., *LoginController*), which exposes two public methods (i.e., *loginAsCustomer()* and *logout()*).

```
1  @RequestScoped
2  public class LoginController {
3      private String username;
4      private String pwd;
5      private String toHome = "index?faces-redirect=true";
6
7      @Inject
8      private LoggedUserComponent loggedUser;
9      @Inject
10     private UserDao userDao;
11     @Inject
12     private PasswordManagerComponent pwdManager;
13
14     public String loginAsCustomer() {
15         User user = userDao.login(
16             this.username,
17             pwdManager.encode(this.pwd)
18         );
19
20         if(user != null)
21             this.loggedUser.initUser(user);
22
23         return (user == null) ? "" : this.toHome;
24     }
25
26     public String logout() {
27         this.loggedUser.shutDownUser();
28
29         return this.toHome;
30     }
31 }
```

Listing 5.1: Java™ class definition of *LoginController* in *Flight Manager*, handling the login page. Username and password fields are supporting variables for an HTML form; *userDao* and *pwdManager* are two injected components, representing dependency relationships, while *loggedUser* is initialised only in the case of successful authentication; the *loginAsCustomer()* method uses both dependencies for applying the right query on the database. Only in the case of authentication, it redirects the end-user to the *Home* page, returning a string (i.e., *toHome* attribute) written in the JSF syntax.

5.2 Problem Formulation

While Application Servers are designed so as to serve multiple requests to a plethora of different concurrent clients, a human (not bot) end-user is, theoretically, only able to generate one request at a time. Under this assumption, it can be stated that:

- a DI container will manage at most one request at time for each client, inside a single *session* context;
- each *request* context managed by the container is related to a specific use case, followed by the end-user.

Also considering that a client advances concurrently on two distinct use cases, in the same moment (e.g., opening two different tabs in a browser within a single user session), alternating click actions; it can be confidently supposed that human reaction time is some order of magnitude higher than a request resolution. In so doing, the Application Server always terminates the processing of a request before the arrival of the subsequent request from the same user.

For all these reasons, the rest of the dissertation adopts an *intra-session* perspective, accounting in isolation each *session* context, which can be interpreted as an ordered and non overlapping sequence of *request* contexts related to the same user.

The DI container distinguishes contexts and related contextual instances in two mutually exclusive sets:

- *alive* set, containing the contextual instances related to the components actually stored in-memory. So, an *alive* context is populated only by *alive* instances;
- *dead* set, containing the contextual instances related to the components destroyed by the container, thus releasing the memory. So, a *dead* context is populated only by *dead* instances.

Besides, the DI container has to discriminate contexts and contextual instances according to components execution impact inside the use case, considering two types:

- *active* instances, the visible and referrable components inside the same HTTP request. So, an *active* context manages and intermediates *active* inner instances;
- *inactive* instances, the components *alive* but not *active.* So, an *inactive* context hosts only unreachable instances.

From the *intra-session* perspective, *request*, *session* and *application* contexts are always related to *active* instances, while *use case* contexts can also be related to *inactive* instances.

To clarify, with a general example, these considerations see the conceptual abstraction of Fig. 5.1 which depicts a simple scenario, extracted from the *Flight Manager* case study (Sect. 4), describing two users (named α and β) concurrently operating in the booking process (i.e., UC8). In particular, S_α represents the *session* context of the user α and, in the same way, S_β the *session* context of the user β. In this scenario, while β is already logged-in, α completes the authentication process (i.e., UC11) only through and within the second *request* context $UC11_\alpha$, writing in the *session* context a data object related to its logged user instance. Both users, along their main *use case* contexts (i.e., $UC8_\alpha$ and $UC8_\beta$), have to read/write shared data (i.e., temporary reservations on flights) from/to the disposed *application* context (i.e., A).

Note that this results in two cross-contexts components couplings: the first one, in an *intra-session* perspective, between $UC11_\alpha$ and $UC8_\alpha$ with S_α as the intermediary context for the *logged user* data; the second one, in an *inter-sessions* perspective, between $UC8_\alpha$ and $UC8_\beta$ with the *application* context as the intermediary for the *shared data*, about temporary reservations, between S_α and S_β.

This scenario includes only two contexts which are always *active* (i.e., A, S_β), while the *session* context S_α becomes *active* only after the first request resulting from the login process of user α (previously, S_α was *inactive* because it was originally set up, in background, by the web browser). The *use case* contexts (i.e., $UC8_\alpha$, $UC8_\beta$) alternate between *active* or *inactive* statuses, in correspondence of performed end-user interactions.

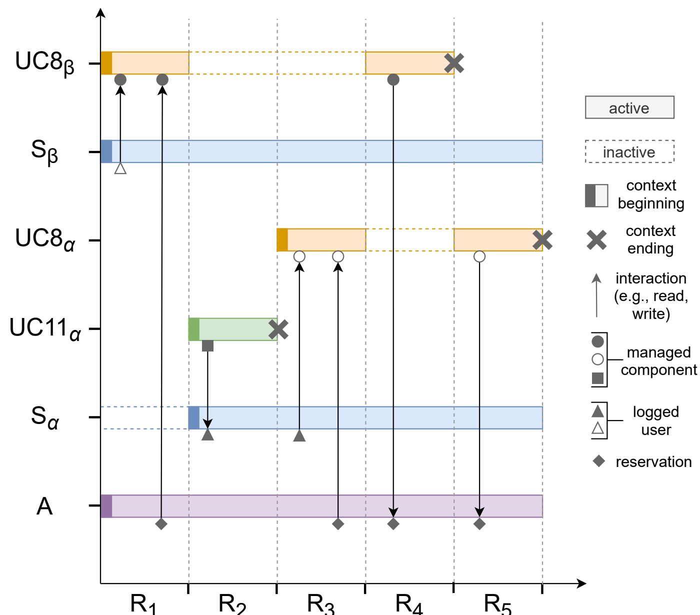

Figure 5.1: *Flight Manager* components couplings scenario, involving two concurrent user sessions. On the *x axis*, *request* contexts (e.g., R_i) are reported in their sequential order. Considering that an HTTP request is the basic context, containing instructions which are applied atomically, the sequence of requests may be considered as a temporal discretisation of server-side computations. On the *y axis*, all the contexts greater than *request* (i.e., A, S_α, S_β, $UC8_\alpha$, and $UC8_\beta$) or special use cases, within an atomic request (i.e., the login $UC11_\alpha$), involved in the scenario are reported. In summary, each Cartesian point describe the state of the context, in the ordinate, within the specific request, in the abscissa.

In general, while *alive* contexts of the same type may coexist within each request, only one context for each built-in type (i.e., *application*, *session*, *use case* or *request*) can be *active*, thus subtending that no overlapping and concurrency is allowed for *active* contextual instances of the same type within a single *request* context. As a consequence, any *active* contextual instance will be able to directly interact only with instances belonging to: its context, a higher level wrapping context, or a lower level wrapped context; respecting the hierarchical organisation fixed by definition. Nevertheless, contextual instances living in disjointed contexts (never active at the same time) may still indirectly interact in a transitive perspective, exploiting *alive* long running instances of higher level contexts.

In the worst case, as depicted in Fig 5.1, the past computation of a *dead* component instance (i.e., $UC11_\alpha$ generating *logged user* data) may affect a future computation of another *alive* and *active* instance (i.e., $UC8_\alpha$ reading the allocated *logged user* data).

In this dissertation, it is considered only the case of a single user in isolation, performing a single use case at a time, excluding residual and rare cases of concurrent requests progressing on two distinct use cases in the same moment, characterising its behaviour under an *intra-session* perspective.

In so doing, there is no overlap between contexts, focusing on faults which may arise only in absence of concurrency among *session* or *use case* contexts.

5.3 Fault Model

Web Applications exploiting frameworks for DI and automated contexts management rely on a DI container to handle components, constraining their contextual instances to visibilities and lifecycles boundaries, hiddenly orchestrating runtime interactions and data sharing.

In so doing, cross-contexts components couplings may be produced in background, implying that past interactions on *dead* contexts may even affect future computations on *alive* and *active* instances.

Inter alia, the DI container should ensure that injected components are automatically selected in the proper type, through a type-safe resolution mechanism. Also in this cases, plenty of possible faults and antipatterns may arise due to error prone programming practices, producing uncontrolled

aggregation of components with different lifetimes and visibility scopes.
The most evident defect may be the wrong definition of the component context, assigning a narrower or a wider scope than necessary as well as exchanging an HTTP built-in scope with the *prototype* one or vice versa.

In other cases, errors may be produced by programmatic bugs due to wrong boundary definitions of *use case* contexts (e.g., beginning of a scope is postponed, or the ending of a scope is declared lately) or by the injection of wrong typed components (e.g., wrong type declarations as well as defects within instantiation algorithm implementations subject to programmatic lookup practices[1]).

As a further consideration, most unpredictable faults may be activated under complex combinations of contexts over direct and indirect dependencies, also as a result of advanced dynamic programming practices [106] (e.g., memoization techniques), or as a result of pattern-oriented implementations [41, 102] (e.g., *Strategy Pattern*, *Decorator Pattern*, *Interceptor Pattern*).

As a direct consequence, a lack of design control is unavoidable: components configuration and implementation require to split meta-information on both sides of a dependency relationship.

The most common meta-configuration foresees that every component type specifies its own context, which is applied by default in every injection of its contextual instances. This implies that the *dependent* component is unaware of the lifetimes of its dependencies and, in turn, an *injected dependency* is unaware of components where it is installed. In this scenario, design of injected components need to be tailored on the requirements of dependent ones; but, this strong coupling implies that an *injected dependency* could behave in an unexpected way if misused.

A variant of this arrangement could be to state the context boundaries of any *injected dependency* inline on *dependent* component side. Counterintuitively, this inline approach exacerbates the introduced criticality: the developer should guarantee that any injected component is structured in a context-independent fashion, so as to be able to correctly operate in any possible combination of dependency usages or, in a limit case, should guarantee that all *dependent* components declare the same context for *injected*

[1] Programmatic lookup is a practice consisting in the implementation of custom policies exploiting low-level API (offered by DI frameworks) for resolving dependencies, dynamically, at runtime.

dependencies, which leads back to the previous scenario (i.e., the simplest way to assure that every component instance lives in the same context type consists in annotating the component itself with the required information). Otherwise inconsistency issues may arise.

In Sect. 5.3.1, on the above premises, a fault model conceptualisation is provided, while in Sect. 5.3.2 the fault model is concretised with some examples contextualised in the case study scenario.

5.3.1 Fault Model conceptualisation

For *stateful* Web Applications, based on DI mechanisms and automated contexts management, the proposed fault model identifies four types of fault:

- **vanishing components** (i.e., contextual instances whose lifetime is early expired) produce dependencies over *dead* dangling references. Usually, this type of fault may be caused by contexts narrower than expected or by *use case* contexts prematurely closed by the developer through programmatic practices.

  Vanishing components may lead to data losses failures, such as null pointer exceptions;

- **zombie components** (i.e., still *alive* contextual instances, as residual memory, that should have been *dead*) produce unexpected couplings over time. Usually, this type of fault may be caused by contexts wider than expected or by temporary or wrong programmatic scope extensions.

  Zombie components may lead to memory leakage failures, exhausting available system memory and contributing to software ageing processes;

- **unexpected shared components** (i.e., two or more components simultaneously depending on same contextual instances and operating over their resources) produce race condition scenarios. Usually, this type of fault may be caused by long-lived components (i.e., application or session scoped) reused in several use cases.

  Unexpected shared components may lead to unexpected shared data and concurrent/un-synchronised accesses to methods and attributes;

- **unexpected injected components** (i.e., injections of wrong typed contextual instances) produce behavioural ambiguity on dependent components executions, deviating the expected use case flow and generating a kind of unpredictability. Usually, this type of fault may be caused by implementation defects of programmatic lookup practices, but also by obsolete source of information adopted for driving the flow of a decision algorithm (e.g., a managed component, maintaining data useful for the dynamic injection of other components, which does not update its data at runtime).

  Unexpected injected components may lead to several failures which can vary from fast fails to unrecognisable ones.

Note that the fault model is tailored over salient characteristics of *stateful* managed components and, in so doing, it does not include common faults for MVC patterns or Object-Oriented programming languages, whose literature is rich and exhaustive [9, 75].

These identified four types of fault may be also generated in combination and a same use case may hide more than one concretisation for each type, so exacerbating the challenge in their detection.

5.3.2 Fault Model concretisation

In this Section the four types of fault have been concretised with reference to some significant use cases of the *Flight Manager* case study (see Chapter 4 for further design details), enlightening how decoupling and reuse of *stateful* components may easily induce a lack of design control without proper verification and testing strategies.

While these concrete fault implementations make tangible the problem in a practical scenario, they also support the discussion about the applicability and effectiveness of the proposed methodology presented in Chapter 7.

Vanishing components The first type of fault is hidden within the "Search Flights" use case, identified as UC7, where the *SearchFlightsController*, living in a *session* context, coordinates the whole procedure within the page named *FlightsResults*, starting from the *Home* page and maintaining a state about end-user's search history, last inspected flights, and filters settings.

This page controller injects an instance of *FlightManagerComponent*, living in a *use case* context starting within the query request and ending once the list is no longer needed[2], which is responsible for query executions and data retrieving. The procedure is optimised so as to retrieve only basic information about flights results in the *FlightsResults* page, demanding for a subsequent further query retrieving the whole flight data, in a kind of *lazy loading* technique, in the case of navigation to a *FlightDetails* page.

In the specific case of triggering the "another search" action, directly from the *FlightDetails* page, a *vanishing component* fault is activated: by design, the previous *use case* context is closed, thus the *FlightManagerComponent* living in it is destroyed by the DI container, and no new *use case* contexts are instantiated. Such a case of managed components reuse (i.e., *FlightManagerComponent*), exploring secondary navigation paths rarely traversed by end-users, may lead to fault and eventually also to failure manifestations[3].

The sequence of HTTP requests leading to the fault occurrence is represented in the conceptual abstraction of Fig. 5.2.

[2]Note that the UC7, retrieving flights from the database can be considered as a preceding relationship for the "Book Flight" use case, identified as UC8. So the *use case* context related to *FlightManagerComponent* will be closed when the end-user select a navigation action forwarding to a booking confirmation or cancelling the research. Indeed, when the flights list is visualised, the end-user could select a flight and begin the booking procedure both from the results list page and the details page, thus ending the *use case* context and redirecting to the appropriate page. Alternatively, the end-user could search a new flight through a new query; also this option is doable from both pages.

[3]A manifestation of this failure is not always clear and evident: the *FlightsResult* page, simply, does not show any results if the manged component responsible of data retrieving does not respond. So the end-user may be confused, but with a forced page refresh he can restart the *use case* context; in this way, however, his navigation experience has been damaged.

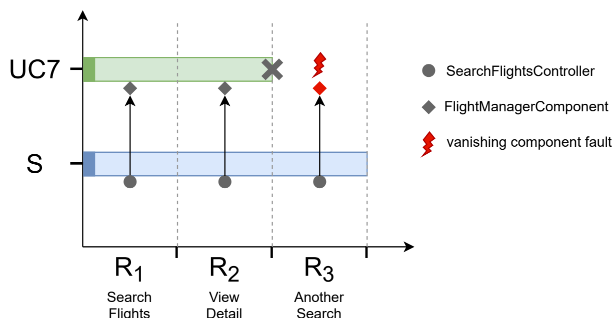

Figure 5.2: Coupling scenario which produces a *vanishing component* fault, both for a visitor and a registered user. In R_1 the end-user starts searching for flights (the *SearchFlightsController* living within the *session* context S depends on the *FlightManagerComponent* living in the *use case* context UC7); in R_2 the end-user views the detail of a specific flight; finally in R_3 the end-user performs a new search (the *use case* context is programmatically closed, then the *SearchFlightsController* tries to invoke the *FlightManagerComponent* living in UC7 which does not exist anymore).

Zombie components The second type of fault is hidden within the "Book Flight" use case, identified as UC8, where two different procedures are provided, distinguishing between visitor and registered users, thus giving some privileges or exploiting *ad hoc* discount strategies. The dedicated page controllers (i.e., *VisitorBookingController* and *RegisteredBookingController*) handling these two different procedures delegate the seats reservation process[4] to their injected instances of *TemporaryReservationComponent*, living in a *prototype* context. Specifically, the *TemporaryReservationComponent* is designed so as to allocate temporary reservations on demand and releasing them just before it is destroyed.

Note that the two page controllers live in different contexts: the *VisitorBookingController*, handling the procedure for visitor users, lives within the *use case* context, while the *RegisteredBookingController*, lives within the

[4]When a flight is selected for booking, the system temporarily reserves a number of seats equal to the declared number of passengers by the end-user. This reservation mechanisms is handled in synergy by the *TemporaryReservationComponent*, which maintains the reservation during the booking procedure, and by *TemporaryReservationRepository*, which takes the total count of reserved seats within the whole *application* context.

session context.

The reuse of *TemporaryReservationComponent*, whose lifecycle is inherited by its injector controller, may produce a *zombie component* fault instance; indeed, while the expected behaviour of the temporary reservation process for seats in the application is correctly achieved for visitor users, the same process for registered users may lead to very insidious faults, difficult to catch because not directly manifested during navigation and not directly affecting final services. Faults occur in a temporal perspective (i.e., as not permanent inconsistencies) in so that allocated temporary reservations of registered users may be held for a time longer than the expected one (i.e., in a correct scenario the system releases the reserved seats when the end-user confirms or aborts the booking procedure, but by design the system releases them only when the *TemporaryReservationComponent* is destroyed, thus when the *session* context ends with the logout of end-user. In this case, a zombie instance of *TemporaryReservationComponent* is generated and the reservation remains in-memory for too long).

The sequence of HTTP requests leading to the fault occurrence is represented in the conceptual abstraction of Fig. 5.3 for a visitor user and Fig. 5.4 for a registered user.

Unexpected shared components The third type of fault is again hidden within the "Search Flights" use case, identified as UC7, and it is due to the reuse of *BillingComponent*, which lives in a *session* context. *BillingComponent* is a managed component with the responsibility of calculating the bill of a booking, as part of this, it also deals with the identification of the fee that should be applied on a flight ticket, which depends on the arrival country (e.g., a ticket for a flight from Italy to Germany has a fee of 19% of its list price, while a fee in the case of return flight has a fee of 22%).

This component is injected at authentication time by the *LoggedUserComponent* (also living in the *session* context), which initialises the *BillingComponent* with the fee value of the country where the end-user lives-in (i.e., retrieving information from its account); in this way, at any time, the *LoggedUserComponent* is able to directly provide the bill calculation (i.e., through a *getHomeCountryFee()* method), acting as a proxy for the *BillingComponent*. In this way, a registered user obtains additional benefits, based

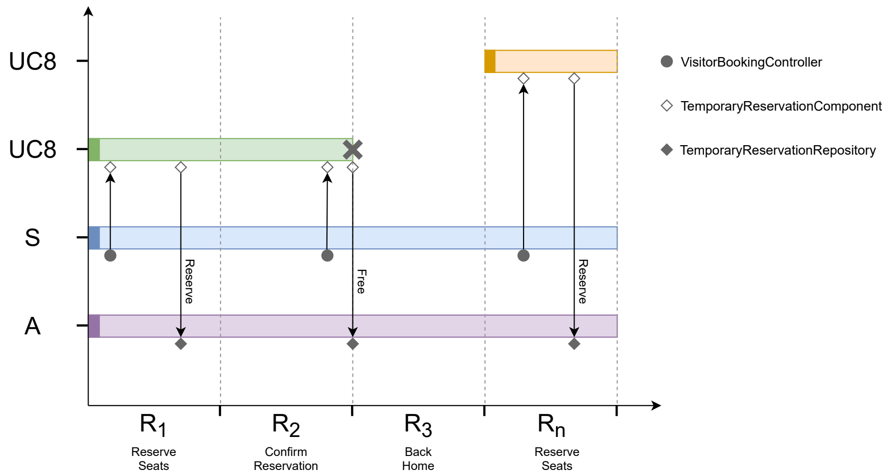

Figure 5.3: Coupling scenario, for a visitor user, which does not produce a *zombie component* fault. *TemporaryReservationComponent* lives within the *use case* context UC8, inheriting from *VisitorBookingController*, and in R_2 it is destroyed by the DI container, thus updating reservation values of *TemporaryReservationRepository* within *application* context A.

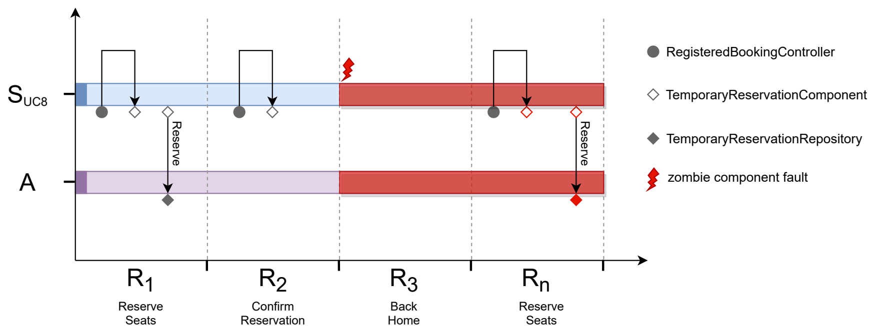

Figure 5.4: Coupling scenario, for a registered user, which produces a *zombie component* fault. *TemporaryReservationComponent* lives within the *session* context S_{UC8}, inheriting from *RegisteredBookingController*, and after R_2 it is always within an active context, thus falsifying reservation values of *TemporaryReservationRepository* within *application* context A.

on the years of affiliation to the platform, when the fee related to his home country is processed.

This configuration may bring the system into an error state, whenever a registered user decides to navigate to the *FlightDetails* page just before buying the ticket, within the use case UC7, for a flight whose destination is a country different from that where he lives. Indeed, the *FlightDetails* page is controlled by *SearchFlightsController* which in turn configures the instance of the *BillingComponent* by setting the country of arrival to the one chosen for the flight; while, in the case of destination within the home country, it directly exploits the *LoggedUserComponent*. These three managed components all live within the same long-running *session* context: they share their contextual instances (i.e., *LoggedUserComponent* and *SearchFlightsController* share the *BillingComponent*). Thus, the last configuration of *BillingComponent* with a foreign country overwrites the initialisation done in the login process by the *LoggedUserComponent*. So, the application enters in an error state[5], which however is not manifested.

Its manifestation may be produced in a subsequent execution of the same use case, if the registered user searches for the return flight to come back to his *home* country. Indeed, navigating again to the *FlightDetails* page, the wrong country is exploited to calculate the fee to apply on the flight (i.e., it is adopted the fee of the previous destination country instead of the *home* country). Obviously, a failure is manifested if and only if the two fees are different.

The sequence of HTTP requests leading to the fault occurrence is represented in the conceptual abstraction of Fig. 5.5.

[5]The error state concerns with the *LoggedUserComponent*, now referencing an instance of the *BillingComponent* which is not configured with its expected country; thus, any subsequent fee computation, based on this information, may be wrong.

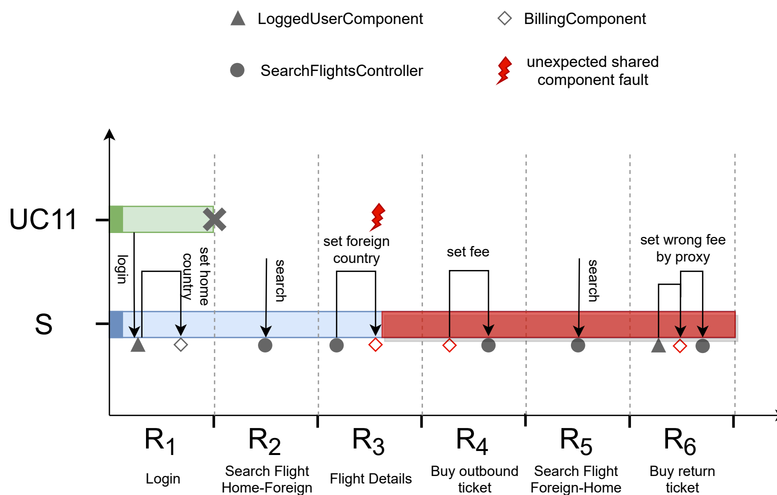

Figure 5.5: Coupling scenario, for a registered user, which produces an *unexpected shared component* fault. *LoggedUserComponent*, *BillingComponent*, and *SearchFlightsController* live within the *session* context S and the data of *BillingComponent* are initialised after the authentication process in R_1. After R_4 the system enters in a latent error state, considering the unexpected sharing of *BillingComponent* contextual instance.

Unexpected injected components The fourth type of fault is again hidden within the "Book Flight" use case, identified as UC8, and directly affects the case of end-users interfacing with *RegisteredBookingController* which is responsible of controlling the *BookingDetails* page. Specifically, the dependencies hierarchy of this managed component involves other three task-specific components, *BillingComponent* living in *session* context, *DiscounterComponent* living in *request* context, and *LoggedUserComponent* living in *session* context.

The "Book Flight" use case has been designed so as to compute in background the final price of a flight ticket and this task is delegated to a chain of responsibility split over the three managed components, mentioned above. The *BillingComponent* is responsible of determining the final price of the booking, applying a country fee on the ticket and asking to the *DiscounterComponent* to determine at runtime if a set of discounts is available for the purchase.

In particular, the *DiscounterComponent* implements a dynamic programmatic lookup algorithm (see List. 5.2 and List. 5.3) for instantiating at runtime the right strategies of discount[6], also basing the decision on some information maintained within the *LoggedUserComponent* (i.e., on the purchasing history of the current registered user).

[6] As described in Sect. 4.3, there are five managed components in the role of *DiscountStrategyComponent*: *i)* the *BigGroupDiscount* which takes into account the number of passengers within a booking and applies a discount if they constitute a group of more than five units, *ii)* the *CrazyWednesdayDiscount* which applies a discount only on Wednesday, *iii)* the *BaseUserDiscount*, *iv)* the *SilverUserDiscount*, and *v)* the *GoldUserDiscount* which apply an incremental discount to registered users with respect to their affiliation levels.

```java
@RequestScoped
public class DiscounterComponent {
    @Inject
    @Any
    protected Instance<DiscountStrategyComponent> discountComponentSrc;

    protected List<DiscountStrategyComponent> activeDiscountStrategies;

    @Inject
    private LoggedUserComponent loggedUserComponent;

    public float apply(Booking booking) {
        float totalDiscount = (float) 0.0;
        // Initialisation of the array activeDiscountStrategies
        initDiscountStrategy(booking);

        for(DiscountStrategyComponent ds : activeDiscountStrategies) {
            totalDiscount += ds.applyDiscount(booking);
        }

        return Util.round(totalDiscount, 2);
    }

    private void initDiscountStrategy(Booking booking) {
        if(activeDiscountStrategies == null
            || activeDiscountStrategies.size() == 0) {
            activeDiscountStrategies = new ArrayList();
            // Programmatic Lookup
            chooseDiscountStrategies(booking);
        }
    }

    private void chooseDiscountStrategies(Booking booking) {
        // See Listing 5.3 for the implementation
    }
}
```

Listing 5.2: Java™ implementation of *DiscounterComponent* living in the *request* context. This task-specific component also depends on *LoggedUserComponent* and dynamically injects contextual instances of a concrete type of *DiscounterStrategyComponent* (i.e., *BigGroupDiscount*, *CrazyWednesdayDiscount*, *BaseUserDiscount*, *SilverUserDiscount*, and *GoldUserDiscount*). The discount strategies selected by the algorithm implemented through the *chooseDiscountStrategies()* method are allocated within an array (i.e., *activeDiscountStrategies*) and applied in concatenation.

```
private void chooseDiscountStrategies(Booking booking){
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(booking.getDate());
    if(calendar.get(Calendar.DAY_OF_WEEK) == Calendar.WEDNESDAY) {
        // CrazyWednesdayDiscount
        activeDiscountStrategies.
            add(discountComponentSrc
                .select(new AnnotationLiteral<CrazyWednesdayDiscount>(){})
                    .get());
    }

    if(booking.getPassengers().size() > 5) {
        // BigGroupDiscount
        activeDiscountStrategies
            .add(discountComponentSrc
                .select(new AnnotationLiteral<BigGroupDiscount>(){})
                    .get());
    }

    if(loggedUserComponent.isLoggedIn()) {
        int userBookingHistory = loggedUserComponent.getHistory();

        if(userBookingHistory > 20) {
            // GoldUserDiscount
            activeDiscountStrategies
                .add(discountComponentSrc
                    .select(new AnnotationLiteral<GoldUserDiscount>(){})
                        .get());
        }
        else if(userBookingHistory > 10) {
            // SilverUserDiscount
            activeDiscountStrategies
                .add(discountComponentSrc
                    .select(new AnnotationLiteral<SilverUserDiscount>(){})
                        .get());
        }
        else if(userBookingHistory > 0) {
            // BaseUserDiscount
            activeDiscountStrategies
                .add(discountComponentSrc
                    .select(new AnnotationLiteral<BaseUserDiscount>(){})
                        .get());
        }
    }
}
```

Listing 5.3: Java™ implementation of *chooseDiscountStrategies()* method of *DiscounterComponent* which performs programmatic lookup. The dynamic discovery and injection of managed components is provided by the CDI framework through the *discountComponentSrc* of type *Instance<DiscountStrategyComponent>*, exposing a *select()* method for programmatic lookup. Note the usage of *LoggedUserComponent* in the business logic of the algorithm.

In this scenario, the fault is not induced by defects within the programmatic lookup implementation, but it may arise when the information owned by *LoggedUserComponent* becomes obsolete and inconsistent during end-users interactions. The *stateful* behaviour of the software promotes a kind of "trust" among managed components, so the *DiscounterComponent* blindly relies on the *LoggedUserComponent* to retrieve information about the purchasing history of the logged user.

Obviously, *stateful* data may be subject to various types of faults which can be caused by classical defects or antipatterns, also as a consequence of previously presented fault types; in this case, the *LoggedUserComponent* retrieves the history of purchasing at instantiation time, but it is not automatically updated when new bookings are accomplished within a same user session. Thus, immediately after the completion of a UC8 use case, *LoggedUserComponent* data may become obsolete, affecting in turn also the programmatic lookup mechanism.

The sequence of HTTP requests leading to the fault occurrence is represented in the conceptual abstraction of Fig. 5.6.

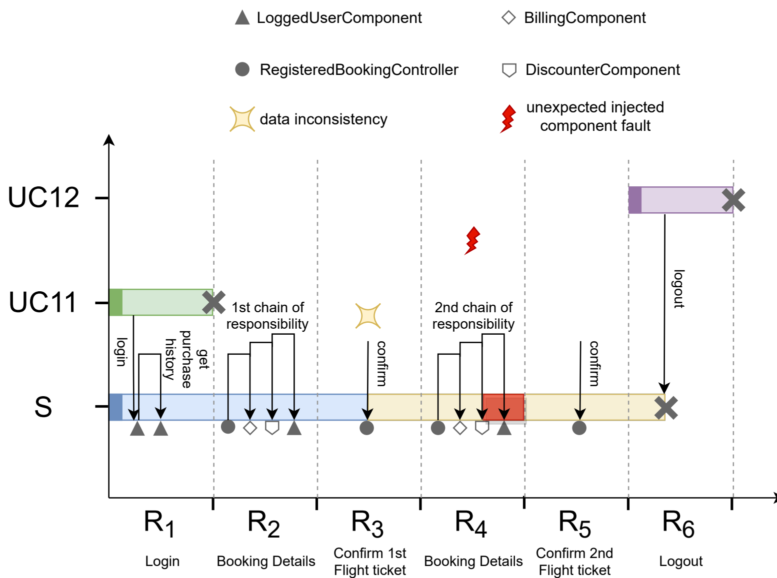

Figure 5.6: Coupling scenario, for a registered user, which produces an *unexpected injected component* fault. Within the *session* context S there are three managed components (i.e., *RegisteredBookingController*, *BillingComponent*, and *LoggedUserComponent*) and by design they also establish a chain of responsibility with *DiscounterComponent* which is injected only when invoked inside a request (i.e., in R_2 and R_4). The programmatic lookup algorithm for dynamic injection, implemented within *DiscounterComponent*, is disrupted after R_2 for the whole end-user session because a data inconsistency is induced on *LoggedUserComponent*.

Chapter 6

Verification of stateful Web Applications

In this Chapter a complete description of the proposed verification methodology is provided, characterising all its stages and introducing, in Sect. 6.1, a formal description of the adopted abstraction for test case generation, named Managed Components Data Flow Graph (mcDFG).

The methodology, inspired by Data Flow Testing approaches, addresses the fault model identified for Web Applications subject to Dependency Injection and automated contexts management, reusing preliminary analysis techniques of the common practice, exploiting custom formalisms based on UML Robustness Diagram and mcDFG, and providing ad hoc coverage criteria.
The main stages of the methodology, presented in Sect. 6.2, can be summarised in: ***i)*** *structural and behavioural preliminary analyses;* ***ii)*** *robustness analysis;* ***iii)*** *robustness diagram decoration;* ***iv)*** *mcDFG generation;* ***v)*** *test case generation.*

6.1 Preamble

So as to support verification of Web Applications exploiting DI and automated contexts management, a methodology focused on the fault model of Sect. 5.3 is proposed.

The methodology also represents a guideline for supporting designers, developers, and testing specialists in the generation of effective test suites at architectural level on how user interactions affect the state and behaviour of managed components. Specifically, since components of a System Under Test (SUT) could interact with - and be *dependent* on the state of - other components, the analysis of the admissible navigation paths generated by end-users assumes a key role during the test case generation process.

In Sect. 6.1.1, a first formalisation of the core abstraction for the test case generation is provided.

6.1.1 The mcDFG abstraction

Coverage of couplings across contexts occurring among components requires a testing approach able to cover the execution paths interconnecting the points where the state of each managed component is defined and used, namely the injection points of in-dependence components and their method invocations, thus capturing the runtime data flow produced by *active* contextual instances.

While, conceptually, the representation of these paths could be modelled through the *Object-Oriented Data Flow Graph* abstraction [108], extending the classical *Data Flow Graph* (DFG) [90]; concretely, this graph representation lacks in the ability of hiding low-level dynamics due to the interposition of DI containers in managing contextual instances (e.g., components proxies, aspect oriented programming techniques). These DFGs, by default, would result in inadequate abstractions both for an explosion of the number of involved edges and nodes within the graph, leading to unfeasible test suites, and for difficulties arising in code interpretation while analysing the application source code (as stated in Sect. 2, low-level DI behaviours are hidden under a simplified syntax through the adoption of meta-information decorations within the source code).

The desired abstraction should jointly depend both on structural characteristics of individual components and on functional characteristics on the

way how views (i.e., web pages) are designed to be navigated during user interactions along use cases. To this end, the set of feasible behaviours of the SUT can be abstracted as a variant of DFG, named *Managed Components Data Flow Graph* (mcDFG), capturing structural and functional perspectives, also taking related works [27,39,47,107] on Data Flow Testing (DFT) as inspiration.

mcDFG The *Managed Components Data Flow Graph* is a directed graph, defined by the following tuple $\langle \mathcal{V}, \mathcal{E}, def, use, Nav, CB \rangle$.
$\mathcal{V}$ is a set of vertices, where each element v represents an atomic set of operations that are always executed as a whole (e.g., components instantiations, components method invocations), similarly to the concept of basic blocks in the classical theory of DFT.
$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, such that $< v_i, v_j > \in \mathcal{E}$ if and only if there is a possible execution where the last operation of v_i can be followed by the first operation of v_j.
Each vertex can be annotated through the *def* and *use* annotation functions, which can be formalised as: $def : \mathcal{V} \rightarrow 2^{mc}$ and $use : \mathcal{V} \rightarrow 2^{mc}$, where mc is the set of all designed managed components within the application and 2^{mc} is the power set of mc. A vertex is decorated with a *def* with respect to a managed component if, in the basic block, the corresponding contextual instance is instantiated (i.e., the DI container performs the injection), rather, a *use* is reported if any method of the related component is invoked. Each vertex is marked with the set of possible *defs* or *uses* occurring in the corresponding basic block, with the assumption that a node accepts either all *defs* or all *uses*.
Finally, since edges could be labelled, two functions have been defined: Nav and CB.
$Nav : \mathcal{E} \rightarrow \{nav\, page\, controller :: sign()\}$, which applies a label to an edge with the indication of page controller method invoked after a navigation action triggered from the User Interface. An edge $< v_i, v_j >$ is annotated with a *nav* label if an end-user interaction produces a transition from v_i to v_j through the page controller method reported within the signature $sign()$.
$CB : \mathcal{E} \rightarrow \{cb\, begin\, use\, case, cb\, end\, use\, case, cb\, end/begin\, use\, case\}$, which applies a *cb* label to an edge respectively when starts, terminates or terminates and immediately starts a *use case* context (modelling the behaviour of a DI container in the management of programmatic contexts).

6.2 The Methodology

The proposed methodology aims at supporting designers and developers in the definition of an effective test suite “fighting” the fault model, presented in Sect. 5.3, for the verification of component-based Web Applications constrained to frameworks for DI and automated contexts management.

The methodology leverages on the *mcDFG* abstraction, which can be (semi-)automatically derived from software specifications emerging from well known artefacts, commonly adopted within agile or ICONIX software development processes [99].

In so doing, the methodology represents an artefact-driven approach characterised by the following stages (which should not to be intended all as mandatory but, in many cases, they can be considered intrinsic to consolidated software development practices):

1. structural and behavioural preliminary analyses;
2. robustness analysis;
3. robustness diagram decoration;
4. *mcDFG* generation;
5. test case generation.

6.2.1 Structural and behavioural preliminary analyses

The first stage of the methodology aims at capturing structural and functional aspects of the under-development application, exploiting documentation artefacts describing main features and expected behaviours, enabling the preliminary design of the *domain model* and the definition of *use cases.*

The domain model design, usually depicted through *UML Class Diagrams*, represents the fundamental of the whole application design process [31], establishing a rigorous vocabulary of the operative domain and a conceptual definition of the entities of interest, which can be continuously refined in compliance to software requirements and development needs.

While the *domain model* captures involved entities and their relationships, *UML Use Case Diagrams* provide a functional perspective of the designed application behaviours, in a graphical and compact format [37], offering a general overview about the application functionalities.

However, these functional diagrams may be integrated with textual documentation written in the shape of the so called *template* formalism [19], detailing for each use case: its unique identifier, a formal description (identifying the final purpose), the involved actors (distinguishing between end-users or system objects), the sets of expected *pre-conditions* and *post-conditions* (from the methodology perspective, these information will help the definition of the oracle verdict), the *main success scenario*, and the alternative execution flows extensions (from the methodology perspective, each flow will help in the understanding of involved components and related method invocations within single sub-step of a use case).

At the same time, through the definition of a *Page Navigation Diagram* [63], characterising the navigation features of use cases, it will be possible to support the identification of involved web pages and to presume underlying transitions method invocations, exploiting hyperlinks.

Note that, these preliminary analysis stages offer a basis for the subsequent robustness analysis stage; providing (in input) a sufficient knowledge to model *boundary*, *controller* and *entity* elements.

6.2.2 Robustness analysis

In ICONIX-based Software Engineering development processes, the robustness analysis is a common practice which demands for the definition of a *UML Robustness Diagram* for each identified use case [98].

This diagram is an artefact that leads to discover and to identify involved actors among use cases, bridging the gap from analysis stage to design stage so as to define relationships among the domain model, the components controlling the business logic and the web pages through its elements:

- *entities*, representing domain model objects;
- *boundaries*, representing web pages;
- *controllers*, representing managed components.

More in depth, *entities* can be extracted from the domain model produced during *structural and behavioural preliminary analyses* stage, with respect to referenced objects within the specific use case.

Boundary elements define a reachability relationship between distinct web pages, further characterising the navigational design of the Web Appli-

cation. In the role of web pages, *boundaries* are the only elements of direct end-user interaction, and in turn, they assume the responsibility of interfaces with *controller* objects.

Controller elements are managed components with the responsibility of controlling a page and/or providing utility methods (e.g., components performing queries on the Database Management System), implementing the whole application business logic. By describing the underneath processes behind an event or a end-user interaction, *controllers* can interact with each others, also redirecting end-users to *boundary* elements and manipulating *entity* objects.

Fig. 6.1 illustrates the UML Robustness Diagram produced after the analysis of the "Book Flight" use case, identified as UC8 in the running case study (see Chapter 4). As a first remark, it should be noted that the diagram is able to represent properly navigability between distinct *boundaries* (i.e., *Home*, *BookingDetails*, and *Confirmation*) also providing an idea of invoked navigation buttons (e.g., click cancel). At the same time, the diagram provides an explanation of what, in broad terms, happens behind a web page, depicting its top level *controller* (e.g., within *BookingDetails* operates a *controller* with responsibilities of initialisation and reservation).

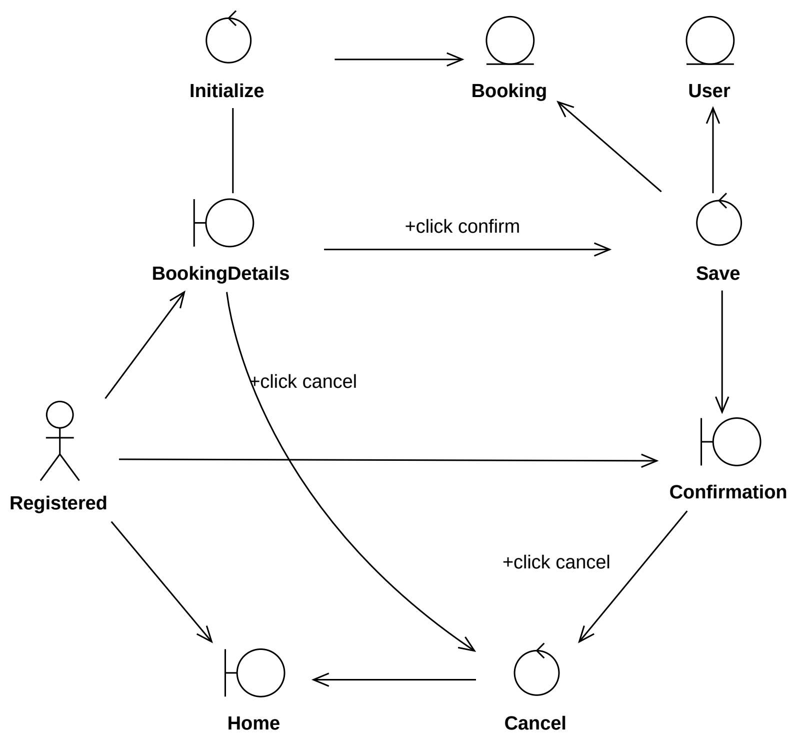

Figure 6.1: UML Robustness Diagram of "Book Flight" use case, identified as UC8 for a registered user.

6.2.3 Robustness Diagram decoration

The UML Robustness Diagram, produced after the *robustness analysis* stage, must be enriched so as to enhance its expressiveness about DI and automated contexts management mechanisms in Web Applications. As a result, the diagram will be enriched generating a finer grained version named *Enriched Robustness Diagram*[1].

The enrichment copes with three enhancement processes:

- page controllers & nav method signatures identification;
- page controllers contexts extraction;
- page controllers dependencies discovery.

The *page controllers & nav method signatures identification* process consists in specifying page controllers related to each *boundary* element (i.e., each web page), decorating the boundary with a UML stereotype related to page controller type (e.g., << *ComponentClass* >>) if relevant, and in outlining all methods bound to client events (i.e., actions explicitly invoked by the end-user or automatically generated by the application) producing explicit routing, asynchronous communications, and partial page renderings. In practical terms, this process marks each *controller* element with an informative label, such as *ComponentClass::method()*, describing the type of the page controller (i.e., *ComponentClass*) followed by the navigational method invoked on its instance (i.e., *method()*).

The *page controllers contexts extraction* process aims to define, for each page controller, its lifecycle and scope. *Controller* elements are decorated with UML stereotypes indicating belonging contexts (e.g., << *request* >>, << *session* >>). Note that this process could be applied *before*, *during*, or *after* coding stage, pursuing different intents.

---

[1]The enrichment stage has the primary intent of decorating the main elements of the diagram with *ad hoc* stereotypes or action edges; e.g., adding a special stereotype (i.e., << *init* >>) to edges related to initialisation page actions performed at page loading or adding *subcall* dashed edges, as described in the procedure. Note that, relationships between *controller* and *entity* elements, for the purpose of the next stage described in Sect. 6.2.4, can be hidden from the enriched version of the diagram.

Applying the process *before* or *during* coding may empower the Robustness Diagram to relieve software developers of making structural choices, also promoting Test-Driven Development (TDD) [8] practices.

Otherwise, applying the process *after* coding enables source code analysis driving an (automated) *a posteriori* enrichment: although reducing the manual effort for developers, this practice may increase the coupling between the SUT and the generated test suite (e.g., defects introduced in the code implementation are tautologically propagated to the diagram).

The *page controllers dependencies discovery* process collects, recursively, all methods invocations within the identified *controller* elements (starting from page controllers), modelling the hierarchy of invocations over their injected contextual instances managed by the DI container (e.g., Data Access Objects operations or utility classes methods).

In practical terms, for each managed component invocation within the page *controller*, this process draws an outgoing dashed edge marked with a sequential number (making explicit the calling order) and adds a new *controller* element, representing the injected component (i.e., the discovered dependency). Each discovered component is decorated with an informative label, such as *ComponentClass::method()*, describing the type of the managed component (i.e., *ComponentClass*) followed by the method invoked on its instance (i.e., *method()*) and stereotyped with its context (as described for the *page controllers contexts extraction* process). The procedure is recursively applied to each discovered dependency, until each *controller* element is fully investigated and decorated.

During the discovery, *use case* contexts boundaries must be determined, decorating dependent *controllers* with: the *UseCaseContext::begin()* label where the use case begins and the *UseCaseContext::end()* label where the use case ends. The information related to contexts management characterise the Robustness Diagram with respect to DI containers behaviour, capturing also the cases of programmatic definition of components lifecycle boundaries.

Undoubtedly, this process requires a great effort when several components are involved but it can be extensively mitigated by automation, implementing source code analysis, searching for all designed managed component methods executions within the involved pages of a use case.

Fig. 6.2 represents the Enriched Robustness Diagram with respect to the diagram in Fig. 6.1. This enriched artefact explicitly depicts, for each *boundary* element relevant for the modelled use case, the indication of its page controller type (e.g., the *BookingDetails* page is controlled by *RegisteredBookingController*) and each *controller* element is decorated with its context and its invoked primary method (e.g., the *RegisteredBookingController* is bound to a *session* context and, on initialisation of the *BookingDetails* page, its *initialize()* method is invoked).

At the same time, an indication of method subcalls (derivable after *page controllers dependencies discovery* process) is reported, if necessary, on dashed edges relationships among *controller* entities (e.g., the same *RegisteredBookingController* includes in its *initialize()* method, a sequence of subcalls related to other managed components, such as *TemporaryReservationComponent* or *BillingComponent*).

6.2.4 Managed Components DFG generation

The Enriched Robustness Diagram, produced after the *robustness diagram decoration* stage, enables the construction of a *mcDFG*, whose syntax and semantics are described in Sect. 6.1.1. This graph abstraction may be used as the main artefact for supporting test case generation in the verification of end-user interactions impact on the application state.

While in classic Data Flow Testing techniques, a DFG exploits coverage criteria to identify testing paths, representing sequences of basic block thus driving the sensitisation of parameters, the *mcDFG* emulates this behaviour combining the architectural perspective, exploiting redefined concepts of *def* and *use* to enlighten components dependency hierarchies, with the navigational perspective, driven by use cases and end-users choices.

In order to identify *defs* and *uses* within the *mcDFG*, the generation process retrieves some useful information about managed components: dependency hierarchies, associated contexts and chain of methods invocations (in response to User Interface events). Considering that a *def* is associated to a contextual instance creation (managed by the DI container) while a *use* is associated to a method invocation of the related component, the generation process requires to “plumb” the overall Enriched Robustness Diagram exploring all feasible navigation paths in order to label basic blocks of the *mcDFG*.

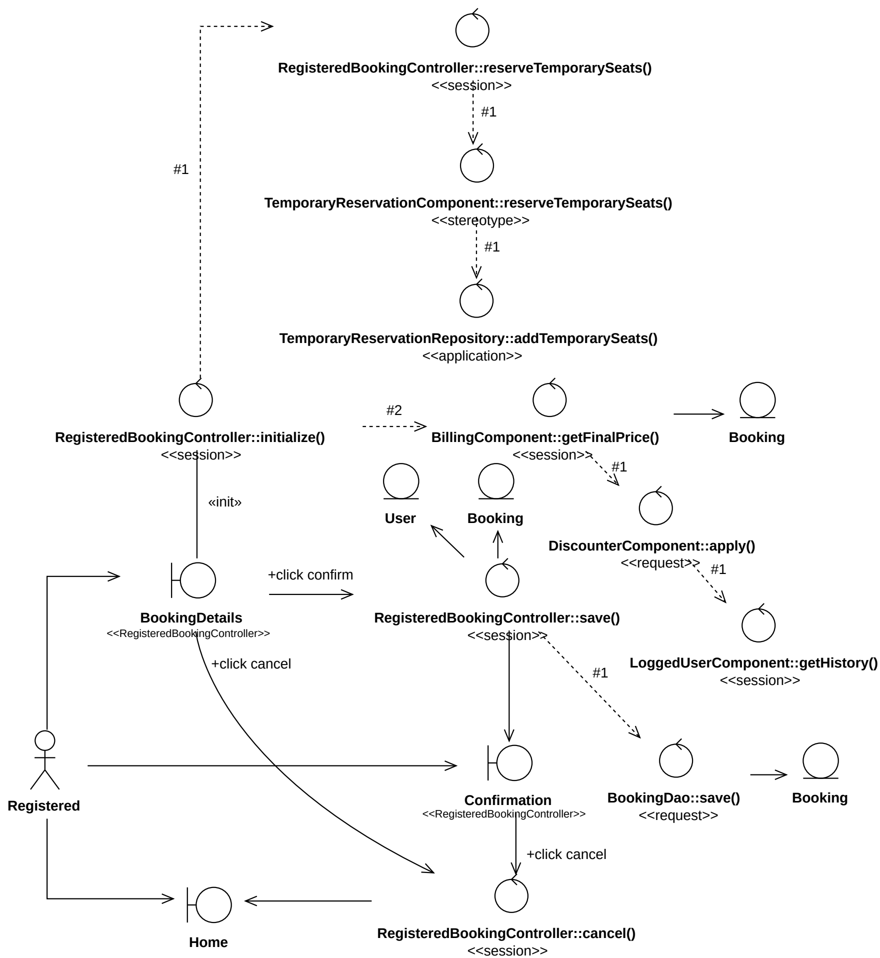

Figure 6.2: Enriched UML Robustness Diagram of “Book Flight” use case, identified as UC8 for a registered user.

Specifically, starting from the *boundary* element related to the use case starting page[2], creations and destructions of all page controllers (encountered during a visit) are identified, in accordance with their contexts boundaries. The same considerations must be recursively reiterated over dependency hierarchies, retrievable through a straightforward analysis performed, again, over the Enriched Robustness Diagram, registering all triggered actions over their functions.

The dependency hierarchy of a managed component c of type *ComponentClass*, can be recursively defined as the set H_c:

$$H_c = ID(c) \bigcup_{\forall \zeta \in ID(c)} H_\zeta$$

where $ID(c)$, abbreviation for Invoked Dependencies, is the set containing all the *controllers* (iterated with the ζ variable) directly invoked by the *controller c*.

Edges of the *mcDFG* are decorated with User Interface events, leading to navigation actions, or control information about *use case* context boundaries (i.e., *cb begin use case*, *cb end use case* or *cb end/begin use case*)[3].

The algorithm producing in output the *mcDFG* is composed by two routines (see Alg. 1 in Sect. A.1 for details):

i) the first routine transforms the Enriched Robustness Diagram, here indicated for brevity with the ERD acronym, in a temporary graph representation named *emcDFG* which can be considered as an expanded version of the final *mcDFG*, containing additional information specific for the algorithm itself[4];

ii) the second routine, reduces the *emcDFG* in the final *mcDFG*, removing unnecessary information, also merging nodes and edges where required.

[2]For the sake of simplicity, this dissertation is discussed over *use cases* owning a single starting point; but the methodology is suitable for the case of multiple starting points too.

[3]From the intra-session perspective, the *mcDFG* does not need to explicitly represent context boundaries for the other scopes. Indeed, *application* and *session* contexts are considered as already initialised (i.e., usually, the login use case, opening a *session* context, precedes other authorisation-based use cases), conversely a *request* context is strictly related to each performed request.

[4]The *emcDGF* is defined as the *mcDFG* through the tuple $\langle \mathcal{V}, \mathcal{E}, def, use, Nav, CB \rangle$, with the addition of the state $\sigma := \langle page, ctrl, ctx, mc, def, use \rangle$ for each node.

The first routine (see Algs. 2, 3, 4, 5, 6 in Sect. A.1 for details) is based on the definition of a state σ for each node of the *emcDFG*, during the construction.

The state is defined as the tuple: $\sigma := \langle page, ctrl, ctx, mc, def, use \rangle$, where *page* is the current page (i.e., initialised with the starting *boundary* element of the ERD), *ctrl* is the label of the higher-level ERD controller responsible for the invoked method, *ctx* is the set of current *active* contexts, *mc* is the set of all alive managed components, *def* is a definition of a managed component, and *use* is a single use of a managed component. Besides, a state σ cannot contain at the same time both a *def* and a *use*, while it may be not associated to any *def* or *use*.

In so doing, the state of the Web Application can be abstracted through the values maintained within σ; thus, the general idea of the first routine, surrounding the transformation process, is to visit the ERD starting from its starting pages (i.e., the *boundary* elements), exploiting any feasible path traversing its edges (following available navigation methods and traversing involved *controller* elements), continuously appending new nodes within the *emcDFG* until reaching a vertex whose state σ is already present in the same *emcDFG* graph (and in this case, a loop is generated and the ERD path is not further analysed).

Two different sub-routines have been identified for handling *boundary* elements and *controller* elements of the ERD. The first sub-routine is responsible for building nodes related to pages (without *defs* or *uses*) and for interconnecting navigation edges (which are outgoing edges of the page). The second sub-routine is responsible for defining a sequence of *defs* or *uses*, retrieving information from the *controller* c associated to a *boundary* or from its dependency hierarchy H_c (the graph requires that *defs* precede *uses* within a sequence of nodes), as well as for identifying edges related to contexts.

The termination of the algorithm implemented by the first routine is assured by the definition of the state σ itself in so that the recursive call of sub-routines exploiting single paths is terminated whenever a preexisting node is reached (i.e., with a state σ equals to the computational one); considering that the number of possible states is limited (indeed, the tuple σ may assume values within a limited set), also the number of the *emcDFG*

nodes is limited and the algorithm will terminate.

The second routine[5] aims at reducing the complexity of the *emcDFG*, visiting all its nodes and edges along available paths, applying a set of transformation/reduction rules for merging nodes and edges.

With its execution, superfluous information is removed, maintaining only navigation labels (i.e., derived from the adoption of the Nav function), context boundary labels (i.e., derived from the adoption of the CB function), as well as *def* and *use* markers. The *emcDFG* graph is then simplified by applying transformation/reduction rules over identified patterns.
Specifically:

- a sequence composed by an unlabelled edge preceding a node without *defs* or *uses* is removed, and the outgoing edges (if present) of the removed node are attached to the node preceding the removed one;
- a sequence composed by an unlabelled edge following a node without *defs* or *uses* is removed, and the incoming edges (if present) of the removed node are attached to the node following the removed one;
- a sequence composed by nodes with only *def*, interconnected by unlabelled edges, is transformed in a unique node reporting on it all *defs*, in the traversing order;
- a sequence composed by nodes with only *use*, interconnected by unlabelled edges, is transformed in a unique node reporting on it all *uses*, in the traversing order.

Finally, in order to include within the final *mcDFG* representation the assumption that all the *application* and *session* managed components are alive when a use case starts, a new starting node containing all their *defs* must be appended to the graph[6], interconnecting it to all previous starting nodes[7] through unlabelled edges.

In Fig. 6.3, the *mcDFG* generated starting from the Enriched Robustness

[5] In Alg. 1 (Sect. A.1), the second routine corresponds to the *reduceToMCDFG()* method invocation. Only textual rules are provided to understand its functioning.

[6] In Alg. 1 (Sect. A.1), this task is accomplished by the *appendFirstDefsNodeMCDFG()* method. Only textual rules are provided to understand its functioning.

[7] If the use case modelled by the Enriched Robustness Diagram has more than one starting *boundary* element, the *mcDFG* has the same quantity of starting nodes.

Diagram of Fig. 6.2 is reported. The dark grey node represents the last appended vertex, containing all *defs* associated to managed components of *application* and *session* contexts. Some edges have been labelled with navigation actions (e.g., *nav RegisteredBookingController::save()*) and two of them are coloured in green and depicted with a dashed line to highlight that, when a path traverses them, the use case ends.

When a node is labelled with a *use* related to a managed component, it means that the method (or the methods) invoked is known, although this information is not directly visible in the graph abstraction for readability purposes.

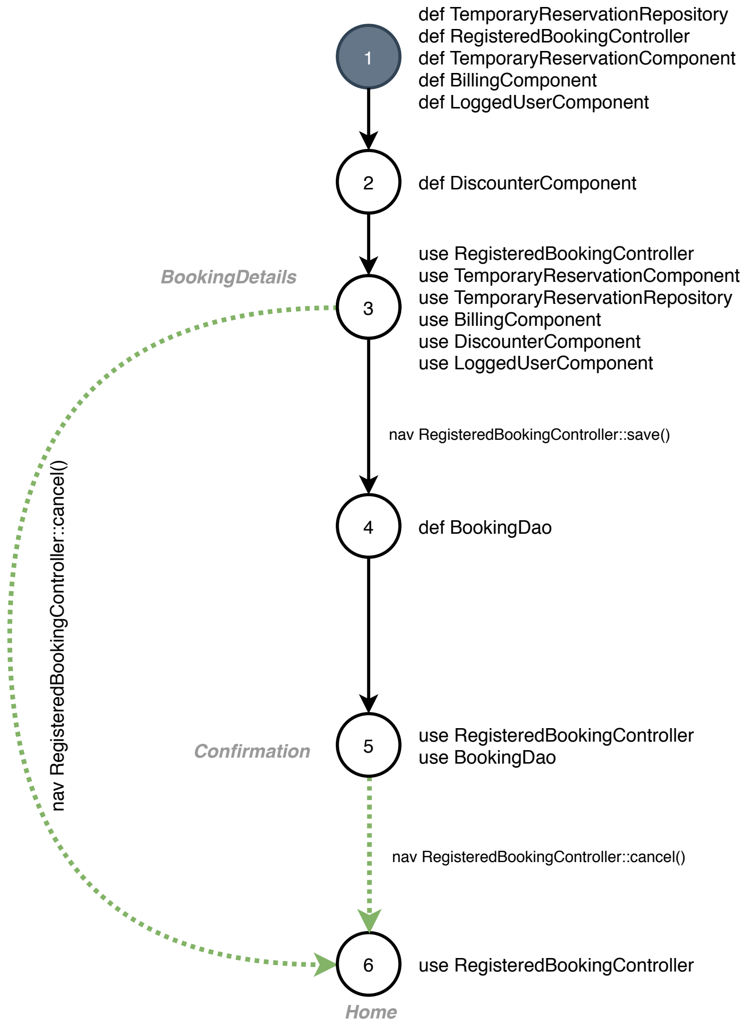

Figure 6.3: Managed Components Data Flow Graph generated from enriched UML Robustness Diagram of Fig. 6.2. The dark grey node represents the vertex decorated with all the *defs* associated to managed components of *application* and *session* contexts. This special node has been appended to the starting page node (i.e., *BookingDetails*), while dashed green edges represent transitions or actions which terminate the use case.

6.2.5 Test Case generation

The *mcDFG* abstraction, produced after the *Managed Components DFG generation* stage, combined with the fault model (described in Sect. 5.3) and *ad hoc* coverage criteria enables the identification of relevant paths leading to the test case generation.

On these premises, the identified test cases will simulate end-user sequences of interaction, aiming to achieve a use case goal to verify both final outcomes and underlying states of the application.

Inspired by Data Flow Testing [90], the *mcDFG* coverage criteria are redefined as follows:

- *All Nodes* criterion verifies that every reachable basic block is tested at least one time, exercising each *def* (i.e., a managed component instantiation) and each *use* (i.e., a managed component method invocation) of a managed component;
- *All Edges* criterion verifies that every edge is tested at least one time, exercising each *nav use* (i.e., each end-user interaction) at least one time;
- *All Paths* criterion verifies that every path is tested at least one time, exercising any possible sequence of *nav uses* (i.e., each feasible combination of end-user interactions derived from navigation design) at least one time;
- *All Defs* criterion verifies that every *def* is tested at least one time, exercising each managed component instantiation, reaching one of its *uses* (i.e., one of component method invocation), without traversing intermediate *defs* of the same component;
- *All Uses* criterion verifies that every *def* is tested one time for each possible *use*, excluding the paths with many intermediate *defs* of the same component related to a method invocation associated to the tested *use*;
- *All DU-Paths* criterion verifies that every *du path* is tested at least one time, exercising every path connecting each *def* of a managed component with all its *uses* (i.e., testing any feasible combination of user interactions from all possible instantiations of each managed component to all possible operations exploiting it).

Redefinition of the concepts of nodes and edges as well as of *defs* and *uses* over the proposed DFG abstraction, with respect to the classical DFT theory, implies that also classical inclusion criteria among coverage criteria must be reconsidered (see Fig. 6.4).

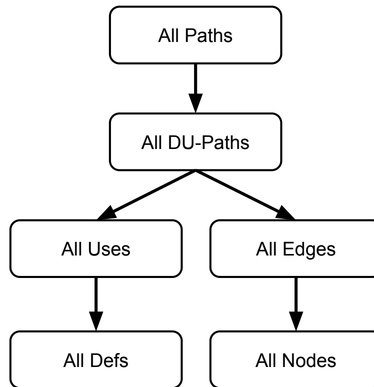

Figure 6.4: Inclusion criteria among coverage criteria for the *mcDFG* abstraction.

Notably, the inclusion criteria strictly related to the graph theory, synthesised by DFG's literature remain in effect [90].
It remains true that:

- *All Paths* includes *All Edges*;
- *All Edges* includes *All Nodes*.

Relationships among criterion involving *defs* and *uses* have to be newly evaluated. Following inclusion criteria have been identified:

- *All Paths* includes *All DU-Paths*.
  This inclusion criterion remains in effect also for the *mcDFG*; indeed, visiting all paths within the graph implies that all edges have been traversed and so all nodes, including associated *defs* and *uses* in any possible path and combination;
- *All DU-Paths* includes *All Edges*.
  This inclusion criterion, which is true also for classical literature of

DFT by the composition of "*All DU-Paths* includes *All Uses*" and, in turn, "*All Uses* includes *All Edges*", is here also valid. Indeed, considering that *All DU-Paths* has to exercise each *def* and each *use* for each managed component in any possible interconnecting path, it is directly evident that each navigation path must be exercised at least one time, thus including *All Edges*. As a remark, every test case begins from a starting node and terminates on a final node, also traversing edges marked with contexts boundary (automatically produced by the DI container or by programmatic management of contexts) or a navigation action;

- *All DU-Paths* includes *All Uses*.
  This inclusion criterion remains in effect also for the *mcDFG* by definition; indeed, visiting all *du path* within the graph implies that for every *def-use* couple at least one path is exercised, thus including *All Uses*;

- *All Uses* includes *All Defs*.
  This inclusion criterion remains in effect also for the *mcDFG* by definition; indeed, visiting at least one *du path* for each *def-use* couple implies also that at least one *du path* is exercised for every *def*, thus including *All Defs*.

Note that inclusion criteria slightly differ from classical theory; this is almost unavoidable in so that while DFT approaches operate under a purely structural perspective, the approach presented in this dissertation operates over a *mcDFG* representing both structural and functional aspects of a Web Application. Specifically, the difference descends from a different semantics of *mcDFG* branches. Indeed, *mcDFG*'s branches does not represent conditional guards (i.e., *p-use* applied on source code variables exploited by predicates) but rather model navigation control choices over managed components, which are determined by user interactions; this invalidates the assumption that each branch necessarily involves at least one *use* as in the classical context.

In so doing, two main classical inclusion criteria decayed: no inclusion relationship can be defined between *All Uses* and *All Edges* neither between *All Uses* and *All Nodes*.

Let N be the number of nodes within the *mcDFG* abstraction, while C the number of distinct managed components, and F the maximum degree of

freedom in choosing a *nav* action within a use case, the complexities theoretical limits are reported in Tab. 6.1.

Complexities of the coverage criteria reflect that of classical theory; more precisely, the complexity of *All Paths* is exponential (if cycles are not considered, elsewhere the complexity is infinite); the complexity of *All Edges* is linear with the number of *mcDFG* edges which can be always less than or equal to the multiplication of the number of *mcDFG* nodes with the maximum number of navigation choices; the complexity of *All Nodes* is linear with the number of the *mcDFG* nodes; the complexity of *All DU-Paths* is dominated by a worst case, leading to the same complexity of *All Paths*, an exponential number of generated test cases; the complexity of *All Uses* is dominated by a worst case where each couple of *mcDFG* nodes has to be exercised, leading to a quadratic complexity; finally, the complexity of *All Defs* is linear with the number of the *mcDFG* nodes multiplied by the maximum number of managed components.

While affordable for *All Defs*, *All Uses*, *All Edges* and *All Nodes* coverage criteria, complexity may become heavy for *All DU Paths*, and *All Paths* criteria applied in scenarios composed by many use cases.

All Paths	**All Edges**	**All Nodes**
$\mathcal{O}(2^N)$	$\mathcal{O}(N \cdot F)$	$\mathcal{O}(N)$

All DU-Paths	**All Uses**	**All Defs**
$\mathcal{O}(2^N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N \cdot C)$

Table 6.1: Complexities of coverage criteria for the *mcDFG* abstraction.

While the above definitions of coverage criteria well fit the verification of a single use case in isolation, the methodology can be generalised to collaborative long-running scenarios where an end-user exercises a sequence of use cases, also sharing some managed component (e.g., a session component) by concatenating execution paths identified in isolation, according to the adopted coverage criterion.

In this dissertation, the verification of a single use case in isolation is named *single run* test case and consists of an ordered sequence of navigation actions within a use case scenario related to a feasible path in the *mcDFG*, while the chaining of k *single runs* is named *k-run*, stressing the execution of ordered sequences spread among several use cases. Considering that *pre-*

cedes relationships may exist among distinct use cases, representing a kind of prerequisite (e.g., to access the functionalities of a restricted area, an application may require to perform an authentication process), the methodology prescribes to apply a *k-run*, complying with *precedes relationships*.

The application of a coverage criterion on a *k-run* must be intended as the generation of a test suite composed by all possible combinations of test cases selected in the *single run* of each use case adopting the same coverage criterion. Thus, the complexity of coverage criteria on the *mcDFG* abstraction under *k-run* strategies, obviously, increases.

Indeed, in theory, the number of test cases to be executed for a *k-run* sequence of k uses cases[8] can be derived as the following product of a sequence:

$$\prod_{i=1}^{k} M_{UC_i}$$

where M_{UC_i} is the number of *single run* tests for the ith use case of the sequence.

Furthermore, an upper-bound is provided as a function of the number of runs. Let UC be the set of all use cases and let M_{UC_i} be the number of *single run* tests for the ith use case UC_i, then $M = \sum_{UC_i \in UC} M_{UC_i}$ is the total number of prescribed *single run* tests.

This implies that M^k is the upper-bound for *k-run* tests, considering all possible sequences of k *single run* tests for each possible sequence of use cases.

Within *k-run* scenarios, for feasibility purposes, *All Edges*, *All DU Paths*, and *All Paths* criteria shouldn't stress each possible combination of use cases but apply some heuristics to detect and group highly coupled use cases in terms of shared components, limiting *k-run* over these cases.[9]

Finally, *mcDFG* diagrams may present cycles (also within a single use case in isolation) which can be handled exploiting boundary-interior strategies [77].

[8] The methodology does not impose a constraint in the chaining of use cases, so a same use case can be present more than one time in the sequence.

[9] A first heuristic may consider to apply *k-runs* test cases only to page controllers with a dependency hierarchy involving managed components operating over the same specific entity of the domain model (e.g., in CRUD operations it is frequent the case of adopting the same DAO contextual instance).

6.2.6 Summary

The proposed methodology has been summarised through the *data flow diagram (dfd)* formalism in Fig. 6.5, where nodes represent single stages of the methodology and edges describe outcomes produced by these processes in terms of artefacts (i.e., *requirements specification*, *UC diagrams and templates*, *domain model*, *page navigation diagram*, *robustness diagram*, *enriched robustness diagram*, and *mcDFG*) and meta-information about managed components (i.e., use cases contexts boundaries, page controllers, invoked methods, *defs*, and *uses*).

Black nodes identify processes inherited by the methodology from structural and behavioural preliminary analyses stages in major software development practices. These processes have not to be considered as an additional effort due to the adoption of the methodology, but as a reuse of artefacts already generated.
Specifically, inherited processes are:

- P1 (i.e., *requirements analysis*), producing in output the software requirements specification useful to derive information about operative domain specification and functional aspects of the application;
- P2 (i.e., *preliminary design*), producing in output a preliminary representation of the domain model, including main entities and their relationships;
- P3 (i.e., *use case analysis*), producing in output a complete description of use cases through diagrams and templates, highlighting required interactions for the primary execution flow and all its alternatives, with fixed pre-conditions and post-conditions;
- P4 (i.e., *preliminary navigational analysis*), producing in output a Page Navigation Diagram;
- P5 (i.e., *robustness analysis*), producing in output a UML Robustness Diagram which captures a preliminary version of involved boundaries, controllers and entities.

White nodes identify light-weight processes that may be applied manually (dashed edges represent the possibility to automate partially or fully these

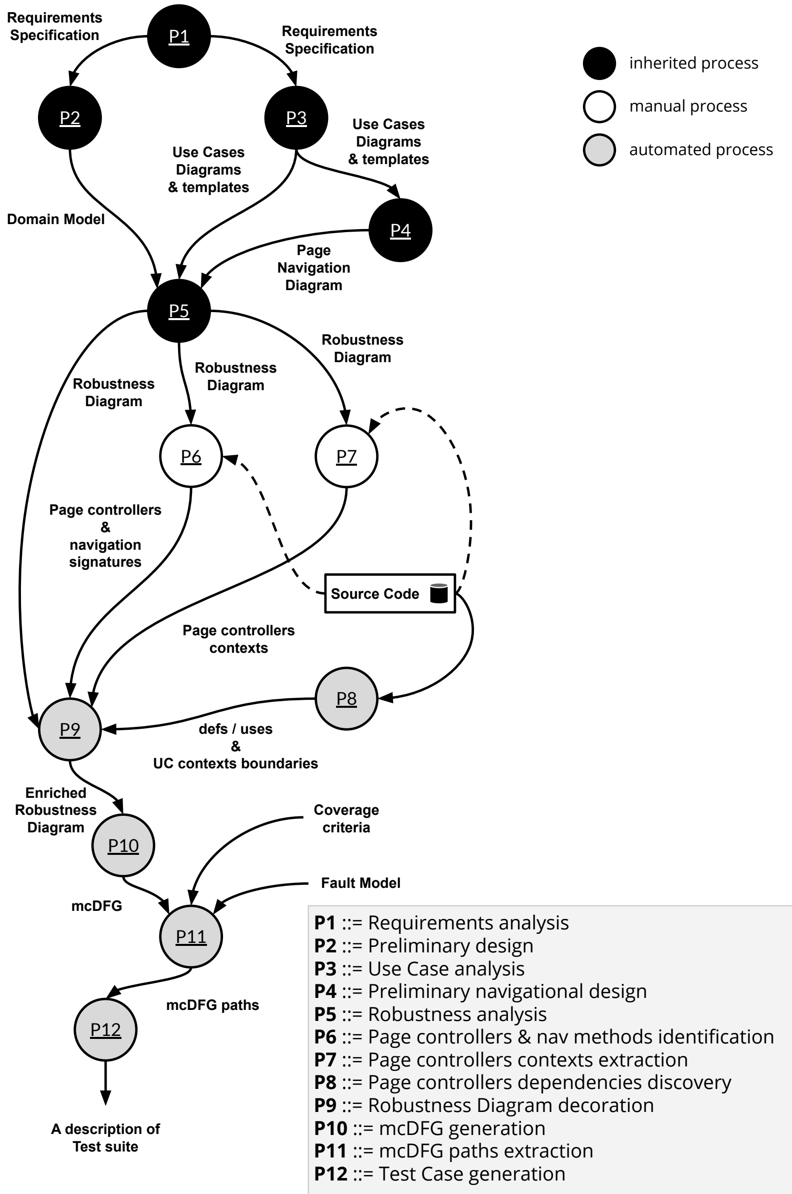

Figure 6.5: Data flow diagram of the proposed methodology.

steps through source codes analysis exploiting *a posteriori* strategies), required to software designers so as to enable the enrichment of the Robustness Diagram.
Specifically, these processes are:

- P6 (i.e., *page controllers & nav methods identification*), identifying page controllers and their methods, raising navigation events;
- P7 (i.e., *page controllers contexts extraction*), detecting managed controllers living contexts.

Finally, grey nodes identify heavy-weight processes that must be fully automated, representing the core of the proposed methodology based on the *mcDFG* abstraction.
Specifically, these processes are:

- P8 (i.e., *page controllers dependencies discovery*), identifying for each page controller the hierarchy of dependencies and their belonging contexts;
- P9 (i.e., *robustness diagram decoration*), producing the enrichment of the Robustness Diagram in order to mark each controller with method invocation labels and related dependency hierarchy subcalls;
- P10 (i.e., *mcDFG generation*), mapping the Enriched Robustness Diagram into a Managed Components Data Flow Graph;
- P11 (i.e., *mcDFG paths extraction*), highlighting paths by applying the fault model and the coverage criteria over the *mcDFG* abstraction;
- P12 (i.e., *test case generation*), mapping the *mcDFG* paths to a descriptive test case, which must be interpreted by developers for applying sensitisation and oracle verdict stages (with respect to designed use cases).

Note that, processes based on automated background source code analysis must be implemented as kinds of adapters for specific technologies and languages, while the overall proposed methodology has general value.

Chapter 7

Discussion

In this Chapter, an evaluation of fault detection capabilities of the presented methodology is discussed with reference to most significant use cases of the running case study (see Chapter 4) and with reference to presented fault model concretisations (see Sect. 5.3.2).

Specifically, in Sect. 7.1, evaluation results as well as costs of coverage criteria applied over generated mcDFG abstractions are reported. In particular, for each use case the minimum number of mcDFG paths is identified for each coverage criterion; further, a discussion about the characteristics of relevant paths and the minimum number of required runs for each fault type concretisation is addressed.
While, in Sect. 7.2, a conclusive discussion supported by qualitative arguments is provided about the required effort for the adoption of the methodology in enterprise-level applications.

7.1 Evaluation of the methodology

In this Section, the applicability of the proposed methodology has been evaluated in order to understand resultant benefits: an evaluation of its fault detection capabilities, applied over the case study (presented in Chapter 4) as well as a discussion of resultant costs are reported in Sect. 7.1.1.

For the sake of concreteness, the methodology has been exercised on the main use cases of *Flight Manager* which concretise and hide all the fault types defined within the fault model (see Sect. 5.3.2 for details), under the assumption of considering the end-user logged in as a customer (i.e., a registered user) or as a simple visitor.

Note that, the provided test case generation strategy identifies a test case as a *mcDFG* path, going from the starting page to the exit page of a specific use case. Such a path, can be translated in an ordered list of navigation actions performed by the end-user, so driving a further test case implementation where sensitisation and oracle verdicts have to be manually defined for the test, relying on information provided by pre-conditions and post-conditions reported within UML Use Case templates.

7.1.1 Fault hunting within the Case Study

The use cases of *Flight Manager* considered for the evaluation are:

- UC7 - "Search Flights";
- UC8 - "Book Flight";
- UC11 - "Login as Customer".

In some scenario, as described for the fault model concretisation, a fault is activated only if the end-user is a registered user, meaning that the authentication process must be performed before the use case, as an implicit *precedes* in the UML Use Cases Diagram. For this reason, the methodology exercises a test case generation through a *k-run* path extraction on the *mcDFG*, where the first use case of the sequence is the "Login as Customer".

A dedicated *mcDFG* diagram has been generated for each investigated use case: UC7 (see Fig. 7.1), UC8 (see Fig. 6.3), and UC11 (see Fig. 7.2).

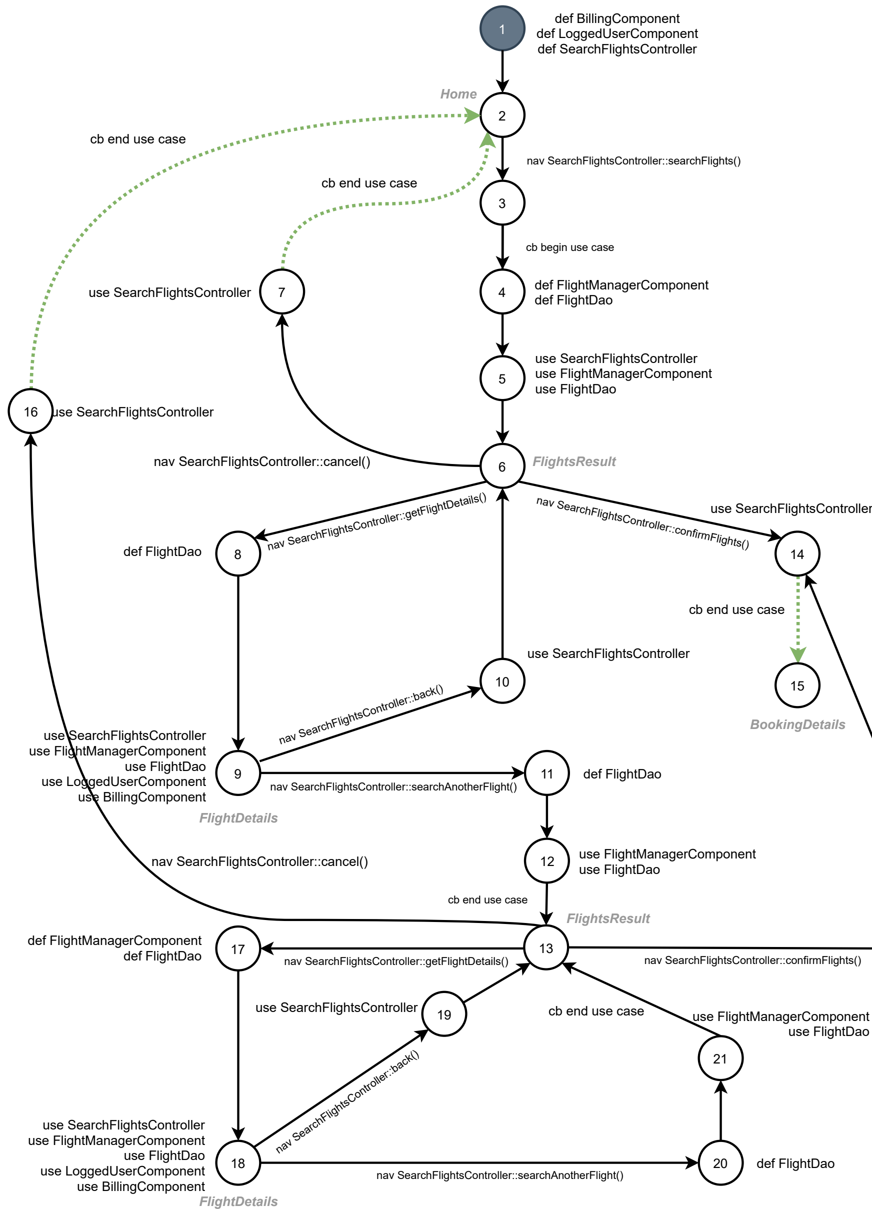

Figure 7.1: Managed Components Data Flow Graph generated from enriched UML Robustness Diagram of Fig. A.4 in Appendix A.

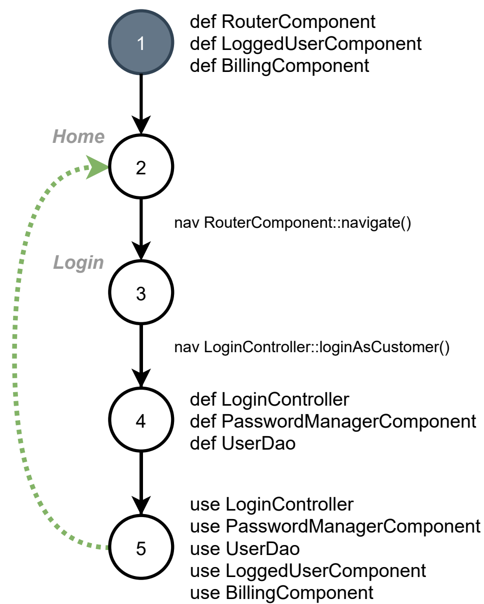

Figure 7.2: Managed Components Data Flow Graph generated from enriched UML Robustness Diagram of Fig. A.2 in Appendix A.

	All Paths	*All Edges*	*All Nodes*	*All DU-Paths*	*All Uses*	*All Defs*
UC7	12	4	3	4	3	3
UC8	2	2	1	2	1	1
UC11	1	1	1	1	1	1

Table 7.1: Minimum number of test cases to be generated for testing the main use cases of the case study in isolation (i.e., executing a *single run*).

The number of *single run* test cases required to satisfy each coverage criterion over each *mcDFG* diagram are reported in the comparison Tab. 7.1. Note that, for UC11 only a test case is sufficient for the complete coverage, thus implying that this use case, substantially, will have a "no impact" in the cost of a sequence of use cases in a *k-run* perspective.

In the following paragraphs, for each fault type concretisation (i.e., *vanishing components*, *zombie components*, *unexpected shared components*, and *unexpected injected components*), qualitative arguments are provided about:

- identification of relevant *mcDFG* paths describing the sequence of navigation actions able to activate the fault, thus leading the generation of effective test cases;
- evaluation of minimum number of runs (i.e., the *k* parameter of a *k-run*) required to detect relevant paths;
- discussion about which coverage criteria guarantee to generate at least one of these paths;
- estimation of the minimum number of generated test cases to ensure that at least one path is relevant.

Vanishing components The first type of fault is hidden within the "Search Flights" use case, identified as UC7, whose *mcDFG* is depicted in Fig. 7.1. The vanishing component of this scenario is represented by the *FlightManagerComponent*, living in the *use case* context. A classical unit testing stage, which exercises it in isolation, is not sufficient to identify the faulty behaviour, caused by an early death of the component itself subject to the programmatic definition of *use case* context boundaries.

The test case generation process, applied over the *mcDFG*, includes the relevant paths for the testing stage through a *single run* (i.e., $k = 1$).

Considering that the fault is caused by a defect in the early ending of the *use case* context, represented in the *mcDFG* by the edge interconnecting nodes n_{12} and n_{13} labelled with *cb end use case*, so paths emphasising the *vanishing component* fault must reach the node n_{13}. In particular, each path exploiting a *use* of *FlightManagerComponent* after that node, may activate the fault after executing the navigation action *nav SearchFlightsController::getFlightDetails()*. In so doing, all the suspected paths must includes in their sequence one of these two cycles:

- $\ldots \to n_{13} \to n_{17} \to n_{18} \to n_{19} \to n_{13} \to \ldots$
- $\ldots \to n_{13} \to n_{17} \to n_{18} \to n_{20} \to n_{21} \to n_{13} \to \ldots$

Thus, each coverage criterion, also *All Nodes* and *All Defs*, is able to define at least one effective test case, capturing the defective behaviour of *FlightManagerComponent*.

The minimum number of test cases required to satisfy coverage criteria are reported in Tab. 7.2, highlighting that a limited quantity of test cases (i.e., in a range between 3 and 12 test cases) is required by any of the criterion, whose values however are far from their theoretical limits.

All Paths	**All Edges**	**All Nodes**
12	4	3

All DU-Paths	**All Uses**	**All Defs**
4	3	3

Table 7.2: Minimum number of test cases required to satisfy coverage criteria for the "Search Flights" use case, identified as UC7, which does not require to be logged in.

Zombie components The second type of fault is hidden in the sequence of "Login as Customer" and "Book Flight" use cases (i.e., in a *k-run* with $k = 2$), identified as UC11 and UC8, whose *mcDFGs* have been presented, respectively, in Fig. 7.2 and Fig. 6.3. This specific concretisation of the fault

cannot be identified by classical unit testing techniques applied over the zombie *TemporaryReservationComponent*. Indeed, this managed component has a *prototype* context, whose implementation is not *a priori* defective but may lead to residual memory if injected by components with extended scopes (e.g., as in case study scenario with *RegisteredBookingController* in *session* context) which blindly rely on its implementation.

On the contrary, by analysing the *mcDFG* paths, capturing components behaviours over sequence of end-user interactions within the use case, it is possible to catch the error produced by *zombie component* fault at runtime.

Considering that the fault is activated whenever a registered end-user exits from the *BookingDetails* page, by cancelling or by confirming the booking, and considering also that each path of the *mcDFG*[1] reaches this page (traversing the node n_3), then each path is able to detect the fault by verifying post-conditions about the number of allocated temporary reservations (with respect to their quantity before the execution of the use case).

Specifically, this use case contains only two alternative paths both effective as test cases:

- **confirmation case**
  $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4 \rightarrow n_5 \rightarrow n_6$

- **cancelling case**
  $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_6$

The minimum number of test cases required to satisfy coverage criteria are reported in Tab. 7.3, highlighting that the number of required test cases is very small; indeed, *All Paths* and *All DU-Paths* criteria are satisfied with only 2 test cases.

Unexpected shared components The third type of fault is hidden in the sequence of "Login as Customer" and "Search Flights" use cases, identified as UC11 and UC7, whose *mcDFGs* has been presented, respectively, in Fig. 7.2 and Fig. 7.1. This specific fault concretises itself whenever a registered user selects a ticket for a flight, indeed on this action an update of

[1] Each path which starts in the node n_1 and arrives in the node n_6.

All Paths	**All Edges**	**All Nodes**
$1 \cdot 2 = 2$	$1 \cdot 2 = 2$	$1 \cdot 1 = 1$

All DU-Paths	**All Uses**	**All Defs**
$1 \cdot 2 = 2$	$1 \cdot 1 = 1$	$1 \cdot 1 = 1$

Table 7.3: Minimum number of test cases required to satisfy coverage criteria, for the *k-run* composed by one execution of the "Login as Customer" use case, identified by UC11, and an execution of the "Book Flight" use case, identified as UC8.

the *BillingComponent* fee is performed, overriding configurations installed at login time.

The above behaviour cannot be identified through classical unit testing techniques applied over the *BillingComponent*, whose instance sharing within *session* context may generate runtime errors, neither it is identifiable with this methodology applied in isolation for UC11 or UC7 (i.e., analysing the *mcDFG* separately).

In these circumstances, only the selection of test cases evaluating collaborative long-running scenarios where a sequence of use cases is exercised becomes effective; thus, evaluating indirect couplings generated among different use cases through shared components. This concrete fault instance can be identified by firstly executing the "Login as Customer" use case and then two times the "Search Flights" use case (i.e., one for search the outbound flight and another for the return flight): the first overwrites the configuration over the *BillingComponent* by entering in the *FlightDetails*, while the second uses the end-user home country fee through *LoggedUserComponent*, acting as a proxy for the same *BillingComponent*, that however has been overwritten.

A *k-run* application of the methodology with $k \geq 3$, according to any coverage criterion, is able to detect the fault concretisation. Knowing that a *k-run* test suite prescribes a number of test cases equal to the product of a sequence of the number of test cases prescribed for each *single run* (related to a single use case among chosen k ones), considering also that the *mcDFG* of the UC11 contains a single path[2], and taking into account costs for the use case UC7 (reported in Tab. 7.2), the number of expected test cases per

[2]The *mcDFG* of UC11, in Fig. 7.2, contains a single path thus implying that the number of prescribed test cases for each coverage criterion is equal to 1.

coverage criterion is reported in Tab. 7.4.

All Paths	**All Edges**	**All Nodes**
$1 \cdot 12 \cdot 12 = 144$	$1 \cdot 4 \cdot 4 = 16$	$1 \cdot 3 \cdot 3 = 9$

All DU-Paths	**All Uses**	**All Defs**
$1 \cdot 4 \cdot 4 = 16$	$1 \cdot 3 \cdot 3 = 9$	$1 \cdot 3 \cdot 3 = 9$

Table 7.4: Minimum number of test cases required to satisfy coverage criteria, for the *k-run* composed by one execution of the "Login as Customer" use case, identified by UC11, and two executions of the "Search Flights" use case, identified as UC7.

Obviously, long-running scenarios may take into account less-costly coverage criteria; indeed in this scenario also *All Nodes*, *All Defs* and *All Uses* are able to detect the fault.

Unexpected injected components The fourth type of fault occurs in the same scenario of the *zombie components* fault, thus it is hidden in the sequence of "Login as Customer" and "Book Flight" use cases, identified as UC11 and UC8, whose *mcDFGs* have been presented, respectively, in Fig. 7.2 and in Fig. 6.3. In general, this type of fault is determined by wrong choices in programmatic lookup practices (i.e., a programmatic injection of component instances, resolving dependencies at runtime) which can be due either to defective algorithms implementations[3] or to runtime hidden errors over the "state" adopted by choosing algorithms. Specifically, this dissertation presented a fault concretisation in Sect. 5.3.2 of this latter case, where the programmatic injection algorithm implemented within the *DiscounterComponent* is correct, but it relies on a potentially obsolete state contained into the *LoggedUserComponent*.

Considering that the fault is activated whenever a registered user performs the UC8 use case a second time, after having already and successfully booked a first flight in the same session, then each path of a *k-run* (with $k \geq 3$) application of the methodology is effective if and only if it traverses

[3]Note that the case of defective implementations of the algorithm logic is less interesting for this research because fault of this type can be easily identified through classical unit testing in isolation.

all nodes of the *mcDFG* of UC11, the n_4 and n_5 nodes of the first run over the *mcDFG* of UC8, and finally the n_3 node in its second run.

The minimum number of test cases required to satisfy coverage criteria are reported in Tab. 7.5, highlighting that the number of required test cases is small for each criterion (also *All Paths* is feasible). This is primarily due to the linearity of the *mcDFGs*, built over use cases designed with less alternative flows[4].

All Paths	**All Edges**	**All Nodes**
$1 \cdot 2 \cdot 2 = 4$	$1 \cdot 2 \cdot 2 = 4$	$1 \cdot 1 \cdot 1 = 1$

All DU-Paths	**All Uses**	**All Defs**
$1 \cdot 2 \cdot 2 = 4$	$1 \cdot 1 \cdot 1 = 1$	$1 \cdot 1 \cdot 1 = 1$

Table 7.5: Minimum number of test cases required to satisfy coverage criteria, for the *k-run* composed by one execution of the “Login as Customer” use case, identified by UC11, and two executions of the “Book Flight” use case, identified as UC8.

In so doing, the simplest coverage criteria in a *3-run* are able to catch the fault, by requiring the definition of at least one effective test case.

7.2 Final discussion

Methodology evaluation shows promising results in “hunting” the concrete fault implementations injected in the *Flight Manager* case study: in Sect. 7.1.1 a characterisation of relevant *mcDFG* paths activating each fault is provided, enlightening in a purely theoretical perspective how the methodology may drive verification stages, prescribing only relevant sequences of end-user interactions (i.e., only feasible sequences of method invocation driven by navigation actions within use cases). These promising results should be better investigated through an actual experimentation, considering the whole software development lifecycle from the design stage to the release stage.

The future adoption of the proposed methodology in concrete operative enterprise-level contexts would demand for a complete cost-benefit analysis,

[4]In particular, UC8 includes a single alternative flow (i.e., cancel) to the main success scenario (i.e., confirm) while the login process does not provides alternatives.

which should consider not only its effectiveness but also the required additional work load in the generation of input design artefacts, and in the implementation of final test cases. For this purpose, a more extensive empirical work on third-party Web Applications, involving professional team of developers and Software Engineers, is advisable. This is out of scope of this dissertation, which provides qualitative arguments basing on the maturated experience over the case study.

On the one hand, in concreteness, the methodology requires to produce only a single type of artefact (i.e., the Enriched Robustness Diagram) useful for the automated generation of the *mcDFG*, which is the core abstraction for the test case generation process. The other mentioned artefacts (i.e., requirements specification, domain model, UML Use Cases Diagrams with templates, Page Navigation Diagram, and UML Robustness Diagrams) have to be considered as standard artefacts of a well disciplined software development process, with the only exception of the robustness analysis stage, which is a typical stage in ICONIX-based practices. Therefore, only the enrichment step of Robustness Diagrams is really a mandatory step, but it is quite clear that it simply prescribes of decorating *boundaries* and *controller* elements with detailed information about managed components, their belonging contexts, and their dependencies hierarchies (for distinguishing among components instantiations and methods invocations). In general, it can be stated that the effort required by the methodology is quite feasible and may be further lightened adopting heuristics for selecting only critical use cases, mainly managed and orchestrated by the underlying DI framework.

On the other hand, as reported in the results of Sect. 7.1.1, the number of test cases to be exercised depends on the adopted coverage criterion. With respect to theoretical limits presented in Sect. 6.2.5 for each coverage criterion, the quantity of effective test cases is lower; indeed, considering that possible end-user interactions and navigation actions are driven by the User Interface and subjected to a specific use case flow, it is very rare reaching the theoretical limits. The *mcDFG* is an abstraction where alternative navigation actions represent alternative execution flows of a use case, then the *mcDFG* is, in most cases, a sparse graph; thus, the number of edges and possible paths within the *mcDFG* is very far from their theoretical maximum. So, the number of expected test cases is usually affordable for *single run* executions of the methodology and may increase only over *k-runs* exploited with heavy coverage criteria.

Chapter 8

Conclusion

This Chapter summarises the contribution of the thesis and discusses avenues for future research.

8.1 Summary of contributions

This dissertation contributes to the area of Model-Based Testing, proposing a methodology for verification of Enterprise Software Architectures with *stateful* components, exploiting Dependency Injection (DI) and automated contexts management.

Specifically, the research addresses the problem of test cases generation for modular Web Applications, realising the Inversion of Control principle through the adoption of DI containers which automatically resolve at runtime components dependencies, also managing components lifecycle, according to client-server paradigm and HTTP fundamentals.

A review of DI frameworks for major programming languages (i.e., C#, Java™, Python™) has been accomplished for comparing common types of context within which the components live and are managed by a DI container, thus enabling the characterisation of a fault model which identifies four specific types of fault, affecting *stateful* applications.

At the core of the methodology a new abstraction, named Managed Components Data Flow Graph (*mcDFG*), has been defined for addressing the fault model by reinterpreting the concepts of *defs* and *uses* of a classical Data Flow Graph, combining structural information (e.g., modelled component

dependencies, components instantiations, components injections, method invocations) with navigational and behavioural aspects of component-based applications (e.g., navigation actions, contexts management).

In so doing, classical coverage criteria of Data Flow Testing have been inherited and reintrepreted to cope with the *mcDFG*, describing inclusion relationships among them and evaluating their theoretical complexities. The application of a coverage criterion supports the automated extraction process of *mcDFG* paths, each one representing a reference description of a single test case. Thus, a test suite may account a use case in isolation as well as a chain of use cases, prescribing the sequence of end-user interactions which must be implemented to exercise the System Under Test in an *end-to-end* testing perspective and within *intra-session* scenarios (i.e., accounting in isolation each *session* context, which can be interpreted as an ordered and non overlapping sequence of *request* contexts related to the same end-user).

The final implementation of a test case is delegated to the developer, who must tailor the test in compliance to the adopted programming language, DI frameworks, and available technologies, manually dealing with sensitisation and oracle verdict stages for concretising pre-conditions and post-conditions designed within UML Use Case Diagrams and templates.

The proposed methodology has been integrated with consolidated practices of software development and interpreted as an artefact-driven approach, leveraging on intermediate abstractions for supporting the automated generation of the *mcDFG*. In so doing, a procedure for building the *mcDFG* abstraction starting from an enriched version of a UML Robustness Diagram has been introduced, extracting page controllers from *boundary* elements, navigation method invocations, belonging contexts of each managed component, and dependencies call hierarchies from *controller* elements.

A qualitative discussion about the applicability of the methodology has been reported for a prototype Web Application, implemented with the Java™ Enterprise Edition ecosystem through Contexts and Dependency Injection (CDI) specification as the DI container and reference framework.

The practical application of the methodology within designed case study scenarios, showed promising results in the capability of deriving effective test suites, from generated *mcDFG* abstractions, for detecting fault type concretisations of the characterised fault model, with acceptable effort and costs for its adoption.

8.2 Directions for future work

Ongoing research activities are pursuing two different perspectives.

In the theoretical perspective, future activities will led the research in addressing also *inter-sessions* scenarios, enabling the joint verification of components living in *session* contexts associated to different end-users. In so doing, it will be possible to consider and detect race conditions scenarios produced by concurrent use case executions led by distinct end-users, interfacing with the Web Application. At the same time, this extension of the methodology may also support the verification of cases of race conditions produced by a same client executing in parallel two or more use cases within a single session (e.g., a human end-user opens two tabs in its web browser, a bot agent or a web-scraper agent "crawls" a sequence of pages).

The fault model will be enriched and accompanied with the identification of common anti-patterns for *stateful* applications, highlighting components configurations which are admissible by design but which may produce unexpected faults at runtime. In so doing, the information provided by the identified anti-patterns will help in reducing or avoiding the lack of design control intrinsically latent within Web Applications where several software components cooperate and maintain server-side a dynamic state during use cases execution.

Furthermore, the proposed methodology could be extended to consider also *stateless* Web Applications (e.g., realising a RESTful architecture) decoupling backend and frontend modules and adopting Inversion of Control within their implementations, thus focusing on managed components modelling client-side a *stateful* behaviour and interfacing server-side endpoints.

In the practical perspective, with the aim of enabling the automation of salient stages of the proposed methodology, a Java™ library will be implemented. The main stages which need for automation are essentially related to the core *mcDFG* abstraction which drives the test case generation; so the construction of the *mcDFG* starting from the enriched version of the UML Robustness Diagram should be automated, implementing the two presented routines, as well as, the test suite generation and selection stages, which should automatically extract the feasible paths from the *mcDFG* for each coverage criterion. In general, also the support offered by source code analysis tools or testing libraries may be useful in pursuing this objective.

Appendix A

Appendix

This Appendix includes the listings (as pseudo-code) of the algorithms, introduced in Sect. 6.2.4, for generating an *mcDFG*, and the set of intermediate artefacts and abstractions[1] produced during the application of the proposed methodology over the *Flight Manager* case study (see Chapter 4), for the four types of fault concretisations, described in Sect. 5.3.2.

[1] For completeness of information, only the artefacts and abstractions which have not been previously reported in the dissertation are here included.

A.1 *mcDFG* generation algorithms

In this Section, the algorithms for generating an *mcDFG* starting from an Enriched Robustness Diagram (described in Sect. 6.2.4) are listed as pseudo-codes.

Algorithm 1: ERD to *mcDFG* Mapper

```
   input  : erd (the Enriched Robustness Diagram)
   output: mcDFG (the Managed Component DFG)
   // 1st routine
1  emcDFG ← doMapERDtoEMCDFG(erd)
   // 2nd routine
2  mcDFG ← reduceToMCDFG(emcDFG)
   // Add a first defs node with all application and session components
3  mcDFG ← appendFirstDefsNodeMCDFG(mcDFG)
4  return mcDFG
```

Algorithm 2: `mapERDtoEMCDFG()`

```
   input  : erd (the Enriched Robustness Diagram)
   output: emcDFG (the extended Managed Component DFG)
   // Global variables initialisation
1  emcDFG with sets V ← ∅ and E ← ∅
   // Allocates global contexts
2  CTX ← {application, session}
   // Retrieves managed components from global contexts,
   // considering them as already alive and thus already defined
3  MC ← extractSessionApplicationMCs()
   // Local variables initialisation
4  SP ← extractStartingPages(erd)
5  while SP is not empty do
6  |  e_tmp ← pull an element from SP
7  |  mapNode(e_tmp, null, null)
8  end
9  return emcDFG
```

Algorithm 3: `mapNode()`

input: e (the analysed ERD element)
input: $\text{currentNode}_{mcDFG}$ (the last mcDFG node, to hook to)
input: edgeLabel_{mcDFG} (the label to apply on interconnecting edge)

```
1  switch getElementType(e) do
2      case BOUNDARY: do
3          mapBoundaryNode(e, currentNode_mcDFG, edgeLabel_mcDFG)
4      end
5      case CONTROLLER: do
6          mapControllerNode(e, currentNode_mcDFG, edgeLabel_mcDFG)
7      end
8  end
```

Algorithm 4: `mapBoundaryNode()`

input: e_b (the current boundary element)
input: $\text{currentNode}_{mcDFG}$ (the last mcDFG node, to hook to)
input: edgeLabel_{mcDFG} (the label to apply on interconnecting edge)

```
1  PAGE ← getBoundaryLabel(e_b)
2  CTRL ← null
3  σ ← ⟨ PAGE, CTRL, CTX, MC, ∅, ∅ ⟩
4  preExistingNode ← findAndGetNode(σ)
5  if preExistingNode is not null then
6      node ← new Node(σ)
7      add node to set V of emcDFG
8      edge ← new Edge(currentNode_mcDFG, edgeLabel_mcDFG, node)
9      add edge to set E of emcDFG
10     E_out ← getOutgoingEdges(e_b) // With following order (init, nav)
11     while E_out is not empty do
12         tmpEdge ← pull an element from E_out
           // Applies the Nav function
13         tmpNavLabel ← getNavEdgeLabel(tmpEdge)
14         mapNode(e_b, node, tmpNavLabel)
15     end
16 end
17 else
18     edge ← new Edge(currentNode_mcDFG, null, preExistingNode)
19     add edge to set E of emcDFG
20 end
```

Algorithm 5: `mapControllerNode()`

```
    input: e_c (the current controller element)
    input: currentNode_mcDFG (the last mcDFG node, to hook to)
    input: edgeLabel_mcDFG (the label to be applied on interconnecting edge, if
           necessary)
 1  CTRL ← getControllerLabel(e_c)
 2  beginnedUseCaseContext ← determineContextBoundaryOpening(MC)
 3  if beginnedUseCaseContext is not null then
 4      add beginnedUseCaseContext to set CTX
        // Applies the CB function
 5      edgeLabel_mcDFG ← getCbEdgeLabel(beginnedUseCaseContext)
 6  end
 7  sortedManagedComponents ← CTRL ⋃ H_CTRL
 8  forall mc_tmp ∈ sortedManagedComponents do
 9      if mc_tmp ∉ MC then
            // def
10          add mc_tmp to MC
11          σ ← ⟨ PAGE, CTRL, CTX, MC, mc_tmp, ∅ ⟩
12          [currentNode_mcDFG,edgeLabel_mcDFG ] ←
              appendNewNodeOrPreExisting(σ, currentNode_mcDFG,
              edgeLabel_mcDFG)
13      end
14      if currentNode_mcDFG is null then
15          return // Termination Rule
16      end
17  end
18  forall mc_tmp ∈ sortedManagedComponents do
        // use
19      σ ← ⟨ PAGE, CTRL, CTX, MC, ∅, mc_tmp ⟩
20      [currentNode_mcDFG,edgeLabel_mcDFG ] ← appendNewNodeOrPreExisting(σ,
          currentNode_mcDFG, edgeLabel_mcDFG)
21      if currentNode_mcDFG is null then
22          return // Termination Rule
23      end
24  end
25  closedUseCaseContext ← determineContextBoundaryClosing(MC)
26  if closedUseCaseContext is not null then
27      remove closedUseCaseContext from set CTX
28      remove all getContextualInstances(closedUseCaseContext) from MC
        // Applies the CB function
29      edgeLabel_mcDFG ← getCbEdgeLabel(closedUseCaseContext)
30  end
31  e_next ← getNextERDelement (e_c)
32  mapNode(e_next, currentNode_mcDFG, edgeLabel_mcDFG)
```

Algorithm 6: `appendNewNodeOrPreExisting()`

```
   input: σ (the status of appending node)
   input: currentNode_mcDFG (the last mcDFG node, to hook to)
   input: edgeLabel_mcDFG (the label to be applied on interconnecting edge, if
          necessary)
 1 preExistingNode ← findAndGetNode(σ)
 2 if preExistingNode is not null then
 3 |   node ← new Node(σ)
 4 |   add node to set V of emcDFG
 5 |   edge ← new Edge(currentNode_mcDFG, edgeLabel_mcDFG, node)
 6 |   add edge to set E of emcDFG
 7 |   currentNode_mcDFG ← node
 8 end
 9 else
10 |   edge ← new Edge(currentNode_mcDFG, null, preExistingNode)
11 |   add edge to set E of emcDFG
12 |   currentNode_mcDFG ← null
13 end
14 edgeLabel_mcDFG ← null
15 return [currentNode_mcDFG,edgeLabel_mcDFG ]
```

A.2 Further artefacts and abstractions

In this Section, the UML Robustness Diagrams and their enriched versions are reported, so as to better understand how some *mcDFG* of Sect. 7.1.1 have been generated starting from these previous artefacts.

Specifically:

- Fig. A.1 depicts the UML Robustness Diagram of “Login as Customer” use case, identified as UC11;
- Fig. A.2 depicts the Enriched UML Robustness Diagram of “Login as Customer” use case, identified as UC11, starting from the artefact of Fig. A.1 and useful for generating the *mcDFG* of Fig. 7.2;
- Fig. A.3 depicts the UML Robustness Diagram of “Search Flights” use case, identified as UC7;
- Fig. A.4 depicts the Enriched UML Robustness Diagram of “Search Flights” use case, identified as UC7, starting from the artefact of Fig. A.3 and useful for generating the *mcDFG* of Fig. 7.1.

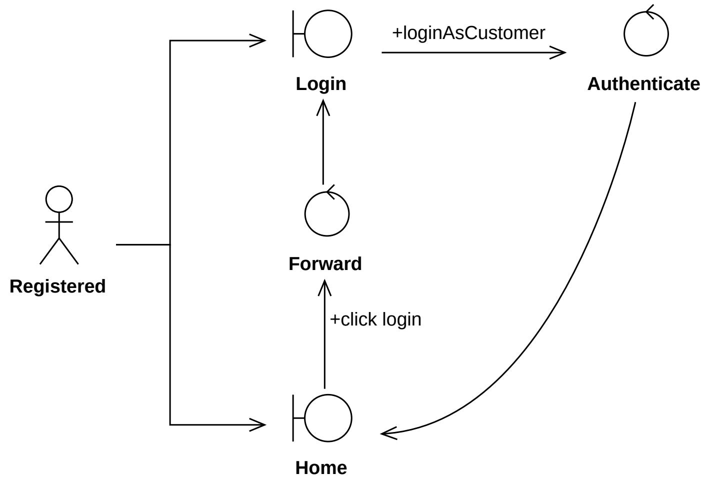

Figure A.1: UML Robustness Diagram of "Login as Customer" use case, identified as UC11.

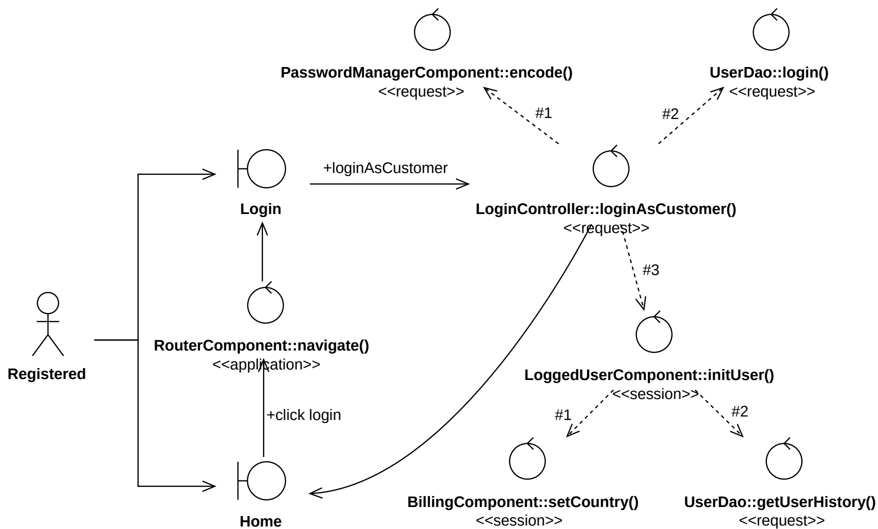

Figure A.2: Enriched UML Robustness Diagram of "Login as Customer" use case, identified as UC11 leading the generation of the *mcDFG* of Fig. 7.2.

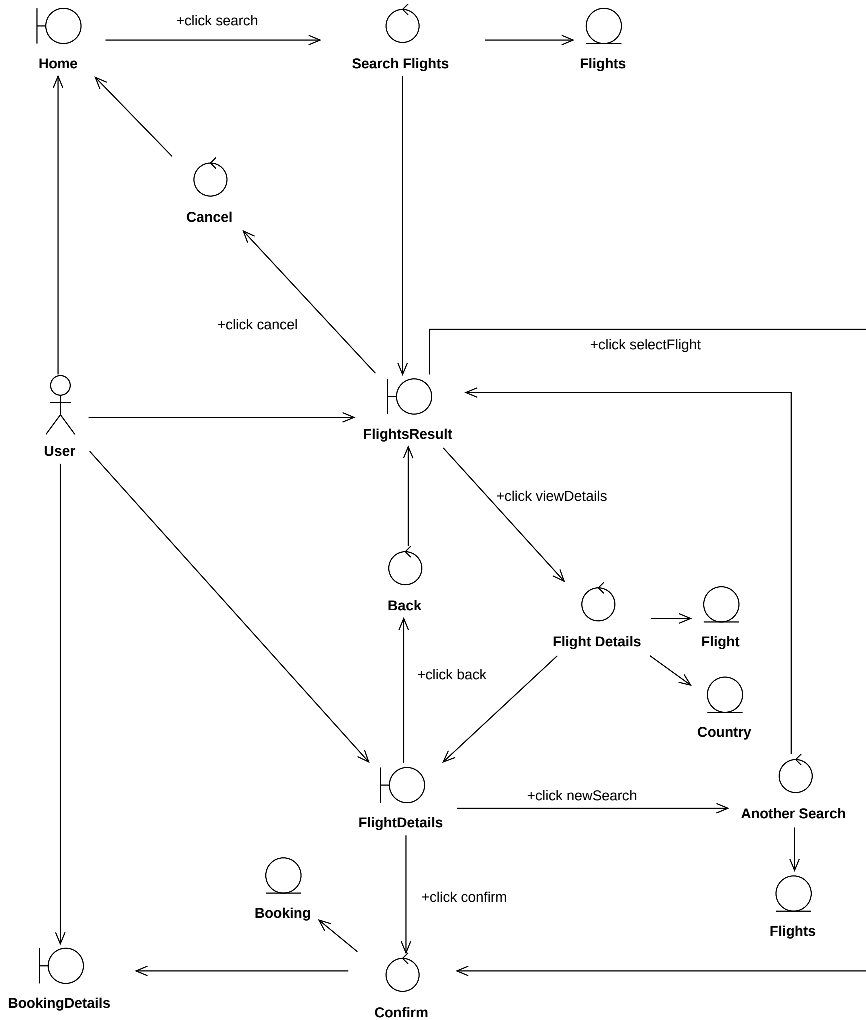

Figure A.3: UML Robustness Diagram of "Search Flights" use case, identified as UC7.

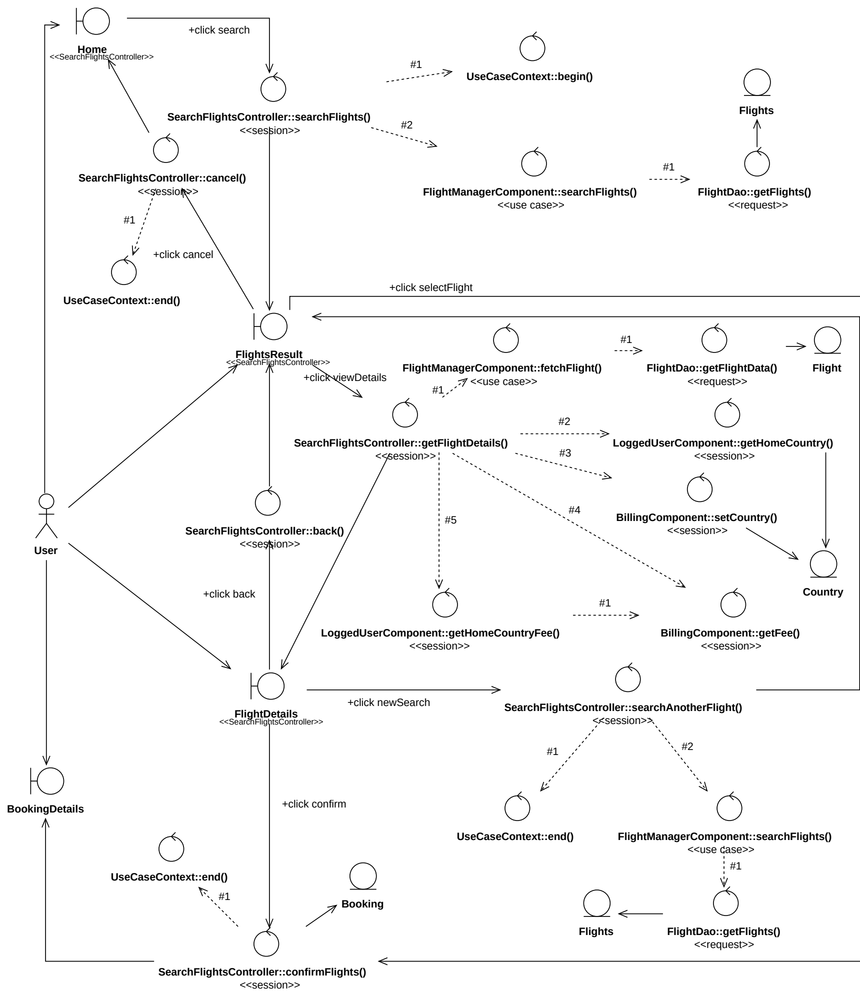

Figure A.4: Enriched UML Robustness Diagram of “Search Flights” use case, identified as UC7 leading the generation of the *mcDFG* of Fig. 7.1.

Appendix B

Publications

Research activity, carried out during the PhD Course, exploited experimentation opportunities offered by several “Research & Development” projects, which I was involved in with research grants, leading to the following publications in International Journals and Conferences.

International Journals

1. **J. Parri**, S. Sampietro, E. Vicario. “Deploying digital twins in a lambda architecture for industry 4.0”, *ERCIM News*, vol. 115, 2018. (Special Issue: Digital Twins)
2. **J. Parri**, F. Patara, S. Sampietro, E. Vicario,. “A framework for Model-Driven Engineering of resilient software-controlled systems”, *Springer Computing Journal*, 2020. [DOI: 10.1007/s00607-020-00841-6]
3. L. Carnevali, A. Fantechi, G. Gori, **J. Parri**, M. Pieralli, S. Sampietro. “A heuristic approach for predictive diagnosis of wheels wear based on a low cost track-side equipment”, *IF-Ingegneria Ferroviaria, Collegio Ingegneri Ferroviari Italiani*, 2021.

International Conferences and Workshops

1. **J. Parri**, F. Patara, S. Sampietro, E. Vicario. “JARVIS, A Hardware/Software Framework for Resilient Industry 4.0 Systems”, in *Proc. of XI International Workshop on Software Engineering for Resilient Systems (SERENE)*, Naples (Italy), 2019. [DOI: 10.1007/978-3-030-30856-8_6]

Bibliography

[1] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[2] D. Alur, J. Crupi, and D. Malks, *Core J2EE patterns: best practices and design strategies.* Prentice Hall Professional, 2003.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[4] Apache Software Foundation, "Apache OpenWebBeans," 2020. [Online]. Available: http://openwebbeans.apache.org/

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[7] A. Barth, "Rfc 6265-http state management mechanism," *Internet Engineering Task Force (IETF)*, pp. 2070–1721, 2011.

[8] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[9] B. Beizer, *Software testing techniques.* Dreamtech Press, 2003.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol–http/1.0," 1996.

[11] G. Booch, *The unified modeling language user guide.* Pearson Education India, 2005.

[12] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise uml for model-based testing," in *Proceedings of the*

3rd international workshop on Advances in model-based testing, 2007, pp. 95–104.

[13] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin, and A. K. Petrenko, "Unitesk test suite architecture," in *International Symposium of Formal Methods Europe*. Springer, 2002, pp. 77–88.

[14] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of at&t pmx/starmail using oats," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[15] E. Burns and R. Kitain, "Jsr 365: Javaserver faces 2.0," 2010. [Online]. Available: https://jcp.org/en/jsr/detail?id=314

[16] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[17] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *arXiv preprint arXiv:2002.12543*, 2020.

[18] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Annual Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[19] A. Cockburn, *Writing effective use cases*. Addison-Wesley Professional, 2000.

[20] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE software*, vol. 13, no. 5, pp. 83–88, 1996.

[21] J. Conallen, "Modeling web application architectures with uml," *Communications of the ACM*, vol. 42, no. 10, pp. 63–70, 1999.

[22] ——, *Building Web applications with UML*. Addison-Wesley Professional, 2003.

[23] S. R. Dalal, A. Jain, G. Patton, M. Rathi, and P. Seymour, "Aetg/sup sm/web: a web based service for automatic efficient test generation from functional requirements," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*. IEEE, 1998, pp. 84–85.

[24] H. Dan and R. M. Hierons, "Conformance testing from message sequence charts," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 279–288.

[25] J. Deacon, "Model-view-controller (mvc) architecture," *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[26] L. Demichiel, "Jsr 317: Javatm persistence 2.0," 2009. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=317

[27] G. Denaro, A. Gorla, and M. Pezzè, “Contextual integration testing of classes,” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 246–260.

[28] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, 2007, pp. 31–36.

[29] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, E.-U. Informaticien, and E. W. Dijkstra, *A discipline of programming*. prentice-hall Englewood Cliffs, 1976, vol. 613924118.

[30] I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing: experience report,” in *Proceedings of the 19th international conference on Software engineering*, 1997, pp. 205–215.

[31] E. J. Evans and E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol–http/1.1,” 1999.

[34] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[35] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[36] ——, “Inversion of control containers and the dependency injection pattern,” 2004.

[37] ——, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[38] ——, “Inversion of control,” *Martin Fowler’s Bliki*, 2005.

[39] P. G. Frankl and E. J. Weyuker, “An applicable family of data flow testing criteria,” *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[40] L. Fuentes-Fernández and A. Vallecillo-Moreno, “An introduction to uml profiles,” *UML and Model Engineering*, vol. 2, no. 6-13, p. 72, 2004.

[41] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[42] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Test generation for graphical user interfaces based on symbolic execution," in *Proceedings of the 3rd international workshop on Automation of software test*, 2008, pp. 33–40.

[43] Google, "Pinject," 2020. [Online]. Available: https://github.com/google/pinject

[44] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 2002.

[45] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.

[46] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 160–170, 1993.

[47] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 154–163, 1994.

[48] M. J. Harrold and M. L. Soffa, "Interprocedual data flow testing," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 158–167, 1989.

[49] F. Ipate and R. Lefticaru, "State-based testing is functional testing," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 55–66.

[50] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[51] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[52] R. Johnson and B. Lee, "Jsr 330: Dependency injection for java," 2009. [Online]. Available: https://jcp.org/en/jsr/detail?id=330

[53] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack *et al.*, "The spring framework-reference documentation," *interface*, vol. 21, p. 27, 2004.

[54] M. Jones, B. Campbell, and C. Mortimore, "Json web token (jwt) profile for oauth 2.0 client authentication and authorization grants," *May-2015.[Online]. Available: https://tools. ietf. org/html/rfc7523*, 2015.

[55] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to uml model-based conformance test generation," in *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 82–91.

[56] M. Katara and A. Kervinen, "Making model-based testing more agile: a use case driven approach," in *Haifa Verification Conference*. Springer, 2006, pp. 219–234.

[57] G. King *et al.*, "Weld-jsr-299 reference implementation jsr-299: The new java standard for dependency injection and contextual lifecycle management," *Nov*, vol. 11, pp. 1–120, 2009.

[58] G. King, L. Red Hat Middleware, and P. F. Draft, "Jsr-299: Contexts and dependency injection for the java ee platform," 2009.

[59] J. C. King, "A new approach to program testing," *ACM Sigplan Notices*, vol. 10, no. 6, pp. 228–233, 1975.

[60] G. Kovács, F. Magyar, and T. Gyimóthy, "Static slicing of java programs," in *University*. Citeseer, 1996.

[61] V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov, "The unitesk approach to designing test suites," *Programming and Computer Software*, vol. 29, no. 6, pp. 310–322, 2003.

[62] C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Yoyoshima, "Design recovery for software testing of object-oriented programs," in *[1993] Proceedings Working Conference on Reverse Engineering*. IEEE, 1993, pp. 202–211.

[63] D. C. Kung, C.-H. Liu, and P. Hsia, "An object-oriented web test model for testing web applications," in *Proceedings First Asia-Pacific Conference on Quality Software*. IEEE, 2000, pp. 111–120.

[64] E. Labs, "Dependency injector," 2020. [Online]. Available: https://python-dependency-injector.ets-labs.org/

[65] L. Larsen and M. J. Harrold, "Slicing object-oriented software," in *Proceedings of IEEE 18th international conference on software engineering*. IEEE, 1996, pp. 495–505.

[66] B. Legeard and M. Utting, "Model-based testing-next generation functional software testing," *SoftwareTech News*, vol. 12, no. 4, pp. 9–18, 2010.

[67] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero, "Control and data flow structural testing criteria for aspect-oriented programs," *Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, 2007.

[68] P. C. Linskey and M. Prud'hommeaux, "An in-depth look at the architecture of an object/relational mapper," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 889–894.

[69] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu, "Structural testing of web applications," in *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. IEEE, 2000, pp. 84–96.

[70] B. Mallo, J. D. McGregor, A. Krishnaswamy, and M. Medikonda, "An extensible program representation for object-oriented software," *ACM Sigplan Notices*, vol. 29, no. 12, pp. 38–47, 1994.

[71] P. D. Manuel and J. AlGhamdi, "A data-centric design for n-tier architecture," *Information Sciences*, vol. 150, no. 3-4, pp. 195–206, 2003.

[72] R. C. Martin, "The dependency inversion principle," *C++ Report*, vol. 8, no. 6, pp. 61–66, 1996.

[73] ——, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, 2000.

[74] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[75] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[76] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[77] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE transactions on software engineering*, vol. 14, no. 6, pp. 868–874, 1988.

[78] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *International Conference on the Unified Modeling Language*. Springer, 1999, pp. 416–429.

[79] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM Sigplan Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[80] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[81] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," in *Pioneers and Their Contributions to Software Engineering*. Springer, 1972, pp. 479–498.

[82] ——, "Information distribution aspects of design methodology," 1971.

[83] R. Pérez-Castillo, I. G.-R. De Guzman, and M. Piattini, "Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems," *Computer Standards & Interfaces*, vol. 33, no. 6, pp. 519–532, 2011.

[84] V. Perrone, L. Mainetti, and P. Paolini, "A uml extension for designing usable user experiences for web applications," *IWWOST. 05*, p. 25, 2005.

[85] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 901–924, 2015.

[86] P. V. P. Pinheiro, A. T. Endo, and A. Simao, "Model-based testing of restful web services using uml protocol state machines," in *Brazilian Workshop on Systematic and Automated Software Testing.* Citeseer, 2013, pp. 1–10.

[87] M. Pollack, R. Evans, A. Seovic, F. Spinazzi, R. Harrop, G. Caprio, and C. Rim, "The spring .net framework reference documentation," 2008.

[88] B. Potter and G. McGraw, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[89] D. R. Prasanna, "Dependency injection," 2009.

[90] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE transactions on software engineering*, no. 4, pp. 367–375, 1985.

[91] *Specification: JSR-346 Contexts and Dependency Injection for the Java EE platform (CDI) ("Specification")*, Red Hat, Inc., April 2014, version: 1.2.

[92] *Specification: JSR-365 Contexts and Dependency Injection for Java 2.0*, Red Hat, Inc., March 2017, version: 2.0.

[93] H. Reza, K. Ogaard, and A. Malge, "A model based testing technique to test web applications using statecharts," in *Fifth International Conference on Information Technology: New Generations (itng 2008).* IEEE, 2008, pp. 183–188.

[94] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* IEEE, 2001, pp. 25–34.

[95] L. Richardson and S. Ruby, *RESTful web services.* " O'Reilly Media, Inc.", 2008.

[96] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks*, vol. 33, p. 18, 2008.

[97] M. A. Rood, "Enterprise architecture: definition, content, and utility," in *Proceedings of 3rd IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises.* IEEE, 1994, pp. 106–111.

[98] D. Rosenberg and K. Scott, *use case driven object modeling with uml.* Springer, 1999.

[99] D. Rosenberg, M. Stephens, and M. Collins-Cope, "Agile development with iconix process," *New York, Editorial Apress*, 2005.

[100] A. Sabot-Durand, "Jsr 314: Contexts and dependency injection for javatm 2.0," 2017. [Online]. Available: https://jcp.org/en/jsr/detail?id=365

[101] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society*, vol. 39, no. 2, p. 25, 2006.

[102] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[103] M. Seemann, *Dependency injection in. NET*. Manning, 2012.

[104] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[105] A. Shatnawi, H. Mili, G. El Boussaidi, A. Boubaker, Y.-G. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif, "Analyzing program dependencies in java ee applications," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 64–74.

[106] M. Sniedovich, *Dynamic programming: foundations and principles*. CRC press, 2010.

[107] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1005–1018, 2003.

[108] A. L. Souter, L. L. Pollock, and D. Hisley, "Inter-class def-use analysis with partial class representations," in *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 1999, pp. 47–56.

[109] V. E. S. Souza, R. Falbo, and G. Guizzardi, "A uml profile for modeling framework-based web information systems," in *12th International Workshop on Exploring Modelling Methods in Systems Analysis and Design EMMSAD*, vol. 782007, 2007, pp. 153–162.

[110] Y. Sun, X. Yang, J. Liu, T. Yu, Z. Xu, Z. Wu, and Z. Chen, "Automatic integration testing through collaboration diagram and logic contracts," in *Journal of Physics: Conference Series*, vol. 1187, no. 4. IOP Publishing, 2019, p. 042043.

[111] R. E. Sweet, "The mesa programming environment," *ACM SIGPLAN Notices*, vol. 20, no. 7, pp. 216–229, 1985.

[112] A. Thomas, "Injector," 2020. [Online]. Available: https://injector.readthedocs.io/en/latest/index.html

[113] S. Tiwari and A. Gupta, "A systematic literature review of use case specifications research," *Information and Software Technology*, vol. 67, pp. 128–158, 2015.

[114] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[115] R. Vanbrabant, *Google Guice: agile lightweight dependency injection framework*. APress, 2008.

[116] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of gui testing using a model-driven approach," in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 9–14.

[117] N. Walkinshaw, M. Roper, and M. Wood, "The java system dependence graph," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2003, pp. 55–64.

[118] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *Genetic and Evolutionary Computation Conference*. Springer, 2004, pp. 1400–1412.

[119] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[120] Y. Wu, D. Pan, and M.-H. Chen, "Techniques of maintaining evolving component-based software," in *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 2000, pp. 236–246.

[121] ——, "Techniques for testing component-based software," in *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2001, pp. 222–232.

[122] S. Yacoub, B. Cukic, and H. H. Ammar, "A scenario-based reliability analysis approach for component-based software," *IEEE transactions on reliability*, vol. 53, no. 4, pp. 465–480, 2004.

[123] J. Zhao, "Applying program dependence analysis to java software," in *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, 1998, pp. 162–169.

[124] ——, "Data-flow-based unit testing of aspect-oriented programs," in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. IEEE, 2003, pp. 188–197.

[125] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Software Engineers Association Xian, China, 2004, pp. 346–351.