UNIVERSITÀ DEGLI STUDI FIRENZE

PhD Program in Smart Computing
Dipartimento di Ingegneria dell'Informazione (DINFO)

Taming Runtime Dependencies across Transient Stateful Components in the Business Logic of Software Architectures

Leonardo Scommegna

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Smart Computing

PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Taming Runtime Dependencies across Transient Stateful Components in the Business Logic of Software Architectures

Leonardo Scommegna

Advisor:

Prof. Enrico Vicario

Head of the PhD Program:

Prof. Stefano Berretti

Evaluation Committee:
Prof. Patrizio Pelliccione, *Gran Sasso Science Institute*
Prof. Ezio Bartocci, *Technische Universität Wien*

XXXV ciclo — March 2023

Acknowledgments

I would like to express my deepest appreciation to my advisor, Professor Enrico Vicario, for his invaluable scientific support and profound belief in my work. I am extremely grateful to my supervisory committee for the valuable advice and expertise given during my Ph.D. study. Additionally, the completion of this dissertation, would not have been possible without my evaluation committee who generously provided constructive criticism and suggestion.

I am also grateful to all the members of the Software Technologies Lab for the friendship and scientific help provided.

Lastly, I would like to acknowledge my family and my girlfriend. Their encouragement and infinite patience have kept my motivation strong throughout the duration of this process.

Abstract

In the engineering of software architectures, transient data need to be managed efficiently. As opposed to long-term data, persisting temporary, and usually unstable, information in a database will result in an overhead of transactions and disk usage. In this case, using in-memory storage is considered the best practice. Concretely, transient data are maintained by stateful components that live concurrently in a layer commonly known as the Business Logic layer.

As a request arrives from the interface, business logic components start a response process during which they dynamically develop dependencies with each other also depending on their current internal state. The resulting configuration allows the implementation of advanced mechanisms also introducing the possibility to make the application behavior dependent on past events.

Dealing with this complexity manually is an error-prone practice, and middleware technologies are usually exploited to manage component dependencies and life cycles. However, relying on third-party frameworks comes with a cost: developers often use them without knowing all the underlying details and this may introduce unexpected behavior in the application.

The problem is further emphasized by the tight relationship that the error propagation phenomenon establishes with the sequence of actions that the user may perform on the interface. At coding time, developers have to consider possible combinations of action that could be performed at runtime to predict how the business logic will evolve. Instead, in case of service failure, they may have to deal with a complex fault removal process due to the intertwined propagation scenario caused by the long sessions of usage.

This dissertation studies how business logic components evolve at runtime and what are the consequences of this. As a first step, it is identified the common technical mechanisms involved in software architectures. Then, the error propagation phenomenon is contextualized in the business logic scenario where components react to requests arriving over time. On top of this, three approaches that face the identified problem with different strategies are presented. In order to identify development faults, it is proposed a model-based testing technique that identifies significative sequences of requests as test cases. To improve the robustness against faults activation and error propagation, this dissertation studies the effectiveness of life cycle management mechanisms as a software micro-rejuvenation strategy and the impact of different life cycle design policies is investigated. As a final step, an instrumentation tool able to observe the business logic evolution is presented, opening the way to strategies of runtime verification.

Contents

Chapter 1

Introduction

In the development of software systems, following an architectural style becomes a crucial practice to achieve a variety of so-called *non-functional* requirements which comprise maintainability, testability, code-readability, and extensibility. To this end, architectural styles guide developers in the modularization of the architecture identifying sub-systems with specific responsibilities that emphasize the separation of concern and promote the encapsulation principle (Martin (2017)).

In presence of non-trivial applications, functional requirements go usually beyond simple CRUD operations, this requires the implementation and management of components that can live for an undetermined period of time during which they interact with each other in order to deliver final functionalities to the client. These mechanisms are usually identified as the *business logic* of the system which is commonly further divided into two sub-modules: the *domain model* and the *application logic*. While the domain logic involves purely the mechanisms affecting the domain, the application logic deals with the application responsibilities managing in addition, the transient state of the business transactions also known as *session state* Fowler (2012).

As the complexity of the business logic increases, implementing manually the dependencies management becomes hard and error-prone. As a good practice, the dependency resolution is supported by frameworks that take care automatically of dependencies instantiation, injection, and destruction thus providing a service identified as *dependency injection and automated life cycle management*.

The Statefulness of business logic components enables the implementation of complex behavior of the system. In particular, it makes the response process dependent not only on the current request but also on previous ones. This is a fundamental aspect when dealing with stateful business transactions however, since the functional behavior of the system is strictly related to the sequence of requests sent to the system, testing and also detecting faults when failures occur, become hard and often unfeasible without proper instruments.

This dissertation studies how components of the business logic react to the various inputs performed by the client to the interface and how, according to this, the session state evolves over time, with a particular focus on the consequences of the overall reliability of the system.

1.1 Contributions

In this dissertation, the implications of business logic managed by frameworks of dependency injection and automated life cycle management are studied under various perspectives with a particular focus on the effects on system reliability.

In particular, as a first step, it is characterized how statically specified dependencies and life cycle specifications impact the dynamic data flow couplings and how this is strictly related to the inputs that the client generates at runtime on the interface.

With this ground, it is identified a set of insidious failure modes caused by common faults in the dependencies configuration and it is consequently proposed a model-based testing methodology aimed to provide in a semi-automated manner, a testing procedure oriented to the identification of the faults of interest through the execution of significant input sequences.

Since business logic components live for an unpredictable period of time in a not so reliable environment (usually stored in-memory) it is plausible to assume that eventually a component will be subject to a kind of software aging phenomenon. This threat could be somehow relieved by the mechanism of life cycle management implemented by the above mentioned frameworks of dependency injection, this dissertation deepened this aspect through an experimental study aimed to define how different policies of life cycle design impact the software aging phenomenon.

Finally, relying on the results of the previous step, it is proposed an automatic procedure able to implement an adaptive software rejuvenation technique relying on a software instrumentation that inspects the evolution of the application logic over time. The instrumentation has also been implemented for JEE web applications and opens the way for reliability techniques combining software rejuvenation and runtime verification strategies.

1.2 Structure of the Dissertation

The rest of the dissertation is organized as follows: in Chapter 2, basic concepts and terminology of software architectures are provided with a particular focus on the business logic and how frameworks support the dependency injection and automated life cycle management of application logic components.

In Chapter 3, the model-based testing methodology is initially provided for generic software systems and then the evaluation is carried on for the specific case of stateful software architectures.

Chapter 4 depicts the experimental investigation of how life cycle management can act as a software micro-rejuvenation strategy identifying the life span and the interactions of the components as the two main causes of aging-related failures.

On the basis of the results obtained, Chapter 5 proposes an adaptive strategy of software aging based on software instrumentation able to conduct both offline and online synchronous monitoring opening a way for a runtime verification strategy aware of the software aging phenomenon.

Finally, conclusions and future research directions are drawn in Chapter 6.

As a side note, a collection of related works are provided in Chapter 3. Although this section is present only in this chapter, the vast majority of scientific papers mentioned, also cover the related work of chapters 4 and 5. In order to avoid too small sections, the literature strictly connected to software aging and runtime verification is integrated into the introduction of the specific chapter.

Chapter 2

Software Architectures and Application Logic

This Chapter provides an overview of software architectures and architectural styles deepening the business logic organization with a particular focus on how dependency injection and automatic life cycle management are implemented by third-party frameworks and how this allows the implementation of advanced yet complex mechanisms.

An introduction to software architectures is provided in Sect. 2.1 while in Sect. 2.2 a more detailed description of the business logic structure is given with an in-depth on how transient information is managed; Sect. 2.3 describes the main strategy that frameworks implement to resolve dependency injection and manage components' life cycle; Sect. 2.4 provides an insight into how the specification of the business logic statically defined results in a complex dynamic behavior evolving over time. Finally, Sect 2.5 defines two software architectural styles in the field of web applications and Sect. 2.6 presents two concrete implementations of them.

2.1 Software Architectures

In the engineering of software systems the *Quality of Service Requirements* (*QoS*) become increasingly important as their complexity increases. This is the case of *enterprise applications* (Fowler (2012)), particular applications that usually involve a massive amount of data managed by multiple users. This implies that an enterprise application should deal with complex tasks like: *i*) data persistence, often exploiting one or more database instances, *ii*) concurrent access to data, due to the fact that various users could interact with the application at the same time and possibly generate race conditions, and *iii*) credential management since it is common to define different types of users with different levels of authority and access. These complexities are further exacerbated by the rise of the Web and Big Data: which on the one hand, they made applications very accessible to users through the browser but, on the other hand, has increased exponentially the number of users and data to manage further highlighting the need to develop scalable systems easy to extend and integrate with other systems.

Although *Functional Requirements* could be achieved even with a *Big Ball of Mud* organization of the application (Foote et al. (1999)), improving non-functional qualities like maintainability, testability and extensibility require following criteria outlining a neat structure and design able to extract from the system a collection of *modules* each of them defining a specific responsibility. The set of modules, relations among them and their properties, is usually called *architecture*.

The main role of software architecture is to support the life cycle of the system making the application easy to understand, develop, maintain and deploy and at the same time, minimize the lifetime cost of the system and maximize the developer productivity.

As in the physical world, software architecture often follows a specific style, to this end, *Garlan & Shaw* (Garlan and Shaw (1994)) define the architectural style as follows:

> *" An architectural style defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. "*

A software architecture that implements a specific style then follows a set of guidelines that indicates how a system should be divided into modules and how they should communicate with each other.

Although *state-of-art* software architectural styles may vary in their details, they have much in common: in particular, they share the same objective, which is the separation of concerns, and they all achieve this separation by dividing the system into layers. The layering technique, as a matter of fact, is one of the most common yet effective strategies to break apart a software system: each layer rests on a lower layer, the higher one uses various services defined by the lower layer while the latter is unaware of the higher layer details. In addition, each layer usually hides its lower layer from the layer above e.g., *layer 3* uses the services provided by *layer 2* which in turn uses services of *layer 1*, but *layer 3* is unaware of *layer 1*.

One of the most popular architectural styles is the *Three-Layers Architecture* (also called 3-tier architecture) (Brown et al. (2003); Fowler (2012)) which identifies three primary layers: *Presentation*, *Business* and *Data Source* (see Fig. 2.1 for a graphical representation). The Presentation Layer implements the logic that handles the interactions between the client (human or another software entity) and the system. This can range from managing commands sent from a command line interface (CLI) to processing interactions performed on a Graphical User Interface (GUI) or a browser.

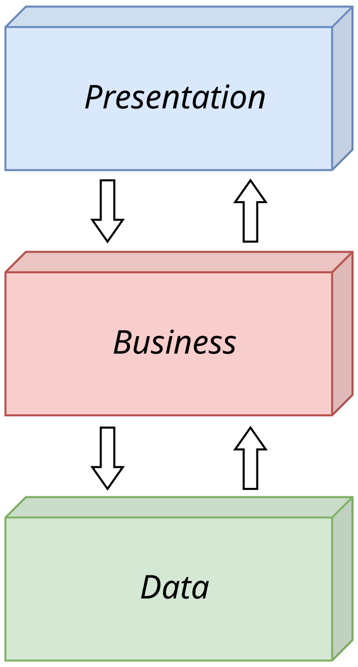


Figure 2.1: The Three-Layers architecture.

The Data Source Layer has the responsibility to communicate with other systems providing services to the application, these systems could be messaging systems, other applications etc...however, in enterprise applications the vast majority of the logic in the Data Source Layer is about the database responsible for storing the persistent data of the application.

Finally, the Business Layer involves all the specific logic that the application needs to achieve its main tasks. It implements the core operations and it is the logic that characterizes the application itself. Usually, the Business Layer maintains a passive behavior meaning that the mechanisms embedded in this layer are triggered by *inputs* sent from the Presentation Layer (i.e., user interactions). More specifically: a client interacts with the system through the interface managed by the Presentation Layer which in turn converts the interaction in an *input* for the Business Layer, once received the input, the Business Layer, with the aim of returning a proper answer to the Presentation Layer, starts an elaborating process involving the business logic of the application and possibly encompassing the lower Layer of Data Source.

It is worth noting that the Three-Layer Architecture identifies a *guideline* in the design of software architectures and many variations are present both regarding the number of layers and their organization. A popular variation in the number of layers, for instance, consists in inserting a *Service Layer* between the Business and the Presentation Layer (Fowler (2012)).

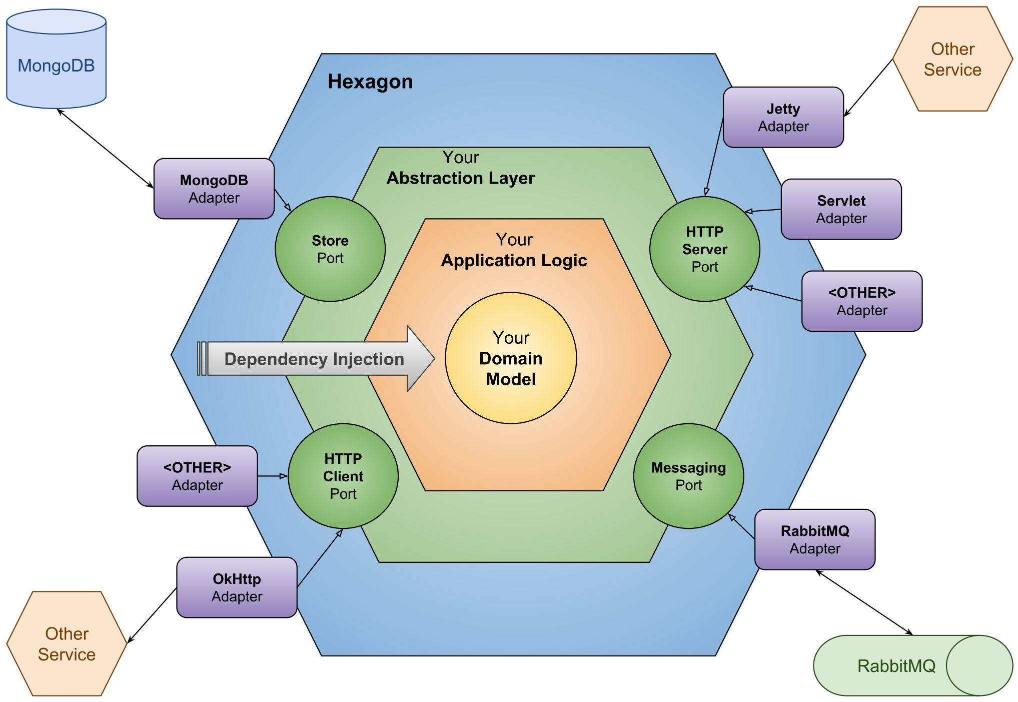


Figure 2.2: The Hexagonal Architecture.

On the other side, architecture like the *Hexagonal Architecture* (in Fig. 2.2), the *Onion Architecture* or the *Clean Architecture* provide a slightly different organization where the business logic is placed at the center and the rest of the system is developed around it in order to prevent its infiltration into the other layers and reaching a high degree of independence from the User Interface, the database and other external agency. This style, especially the *Hexagonal Architecture* one, is gaining more and more popularity, especially with the advent of the micro-service paradigm where the client of the system could be either an end user or another micro-service.

2.2 Application Logic and Session State

The Business logic, no matter the architectural style, represents the core of the architecture, it provides the functionalities and models the domain of the application, however, many designers found it convenient to further distinguish the business into two categories: the *application logic* and the *domain logic* (Fowler (2012); Cockburn (2001)). While the domain logic involves purely the mechanisms affecting the domain (e.g., a discount strategy that may vary depending on the user account), the application logic deals with the application responsibilities (e.g., notifying the user that a discount is available for him before the checkout).

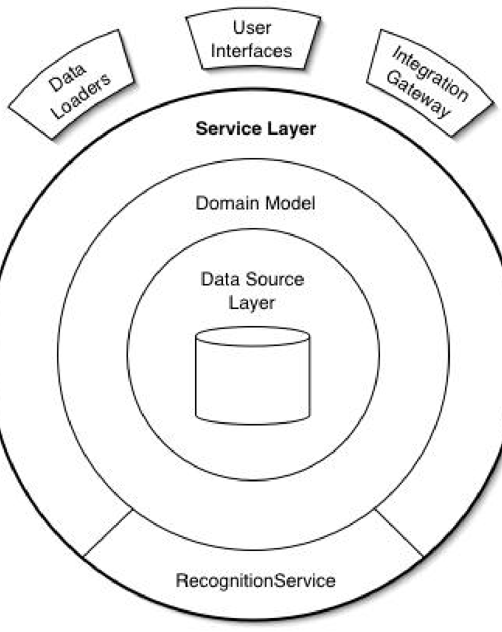


Figure 2.3: Architectural Style with the Service Layer and Domain Model Organization Fowler (2012).

This distinction found its concretization through the realization of an additional layer called *Service Layer*, populated by a set of classes that directly implement the application logic and delegate the execution of the domain logic to an underlying *Domain Model* (Fowler (2012); Buschmann et al. (2007)), an exemplary organization is depicted in Fig 2.3.

In applications with a Graphical User Interface, the *Front Controller* or the *Page Controller pattern* are used in combination with the *Template View Pattern* (Fowler (2012); Buschmann et al. (2007)) where one unique input controller, for the Front Controller case, or one input controller per logical application page, for the Page Controller case, take care of the application logic while the view of the page is responsible to build the page through a template where fields are completed at runtime through the support of controllers. This strategy provides a twofold benefit: on the one hand, it separates the logic from the GUI allowing to build them separately and at the same time, facilitating the testing phase that needs to be performed on classes instead of the interface; on the other hand, it simplifies dramatically the management of dynamic pages i.e., pages where the content depends on external factors like a database query and could look different with each result.

In presence of sufficiently complex systems, the number of classes in the application logic can be substantial, for instance: with large enough applications, it is not rare to have to deal with similar tasks at various points in the application, in this case, a best practice to avoid code duplication and increase the separation of concerns consists in the implementation of *helper classes* (often called *service classes*): classes responsible for the management of common tasks to be shared among multiple controllers (Fowler (2012); Buschmann et al. (2007)).

2.2.1 Stateful Application Logic

Maintaining the application logic stateless, i.e., having controllers and service classes that do not retain state between user interactions, provides several advantages: it maintains the logic simpler, reduces the resources consumption, and, if implemented server-side, allows the processing of other requests from other sessions without race conditions. However, the tight relationship between the application logic and the application use cases often makes it impossible to implement a completely stateless application logic. The main reason is that frequently use cases themselves require their executions to be stateful, in such cases, also the application logic is forced to make the corresponding session (also called *business transactions*) stateful. The state of the application logic is often referred to as *session state* (Fowler (2012); Buschmann et al. (2007)) since it is usually related to a single business transaction and it is not shared among other parallel sessions, however, this is not always the case and there are certain situations where the state involves more than one session.

Even if they both represent a state, the session state and the so-called *record state*, have relevant differences: while the record state identifies the long-term persistent data, namely *record data*, it is visible among multiple sessions and it is usually held in the database, the session state has a transient nature, it is usually stored in-memory (although this is not always the case, as pointed out in Chapter 4) and it needs to be committed to become record data.

To clarify the concept of stateful application logic and emphasize its importance, consider the case of an online marketplace, where one of the most fundamental features is represented by the *shopping cart*. In the domain of e-commerce applications, the typical use case is carried out by users browsing items for sale and occasionally adding to the cart those they want to buy. In this case, the shopping cart represents the session state since the items added to the cart need to be remembered for the user's entire session. Note that such a common feature could not be represented through stateless application logic.

It is worth mentioning that it is not necessary for the session state to be committed and become eventually part of the record state: in the example of the shopping cart, the purchase will be persisted on the database only after the confirmation of the order from the user, before that, it is still possible for the user to abort the operation. This degree of freedom enables the use of the session state also as a convenient and flexible tool in the application logic implementation. Popular uses involve maintaining historical data for a session and improving performances both by exploiting the session state as a kind of in-memory cache and implementing various laziness principles maintaining data into the state postponing time-consuming persistence operations.

2.3 Dependency Injection in the Application Logic

In software engineering, the practice of instantiating and installing dependencies (i.e., *resolution*) on the behalf of the requiring components is commonly known as *Dependency Injection* (DI) and, especially for the application logic development, it is a consolidated pattern that enforces the dependency inversion principle Martin (2000), promotes loosely-coupled classes and improves testability and maintainability. However, it is clear that even in small and medium software architectures, the application logic can easily becomes a complex structure made of various *components*, i.e., controllers and service classes, that develop dependencies on each other.

In this scenario, instantiating all the components and implementing the dependency injection pattern manually results in an error-prone practice. This situation is further exacerbated in presence of procedures requiring a session state: the functionality provided by a component instance may depend on its current state, and so, sharing a stateful dependency among multiple components, may incur in a race condition leading to unexpected side effects.

These aspects of the application logic management are such a common and well-known pitfall that usually the manual dependency injection is avoided and in its place, it is considered best practice to rely on an external participant, often known as *dependency injection container* or *injector*, responsible of creation, sharing and resolution of dependencies which in this case are often referred to as *managed components*. The DI container implementation is ensured by so-called dependency injection frameworks which, due to their relevance, exist for virtually all programming languages e.g., Java, Python, C#, Angular, Vue, React (more details in Sect. 2.5)

2.3.1 Dependency Resolution through Visibility Contexts

Despite the great variety of implementations available, all the Dependency Injection Frameworks provide dependency resolution as their main feature and they even share similar mechanisms to implement it.

Dependency resolution involves three main actors: *i*) the client component *ii*) the required component and *iii*) the dependency injection container. In the specification of the client component (i.e., usually intended as the plain code definition of the component class), a dependency on a specific required component type is declared through meta-information (the type of meta-information strictly depends on the specific framework). In a stateless world, this should be sufficient for the DI container to safely resolve dependencies, however, with stateful application logic, multiple instances of the same required component type might provide a different behavior based on their current state, in this situation, identifying the right instance to inject in the client component becomes fundamental for the functional perspective.

So statefulness, although necessary, introduces an ambiguity between dependencies requirements, specified statically at the component type level, and the actual injection of dependencies, which must be solved at runtime identifying the correct stateful instance among a set of candidates. With the aim of providing an unambiguous dependency resolution mechanism, dependency injection frameworks have introduced the concept of *visibility* and *visibility contexts*.

The visibility relation guides the DI container in the identification of the correct instance to inject into a given instance of the client component and it basically states that a required component instance, "*componentA*", can be injected into the client component instance, "*componentB*", if and only if "*componentA*" is visible for "*componentB*".

To identify efficiently all the visible instances, frameworks implement a criterion relying on the concept of visibility context. In particular, a visibility context defines a "visibility range" also known as *scope*, the components instances associated with a visibility context inherit its visibility range and they are singleton in the context, meaning that no other instances of that component type can be associated with the same visibility context. Instances are associated with a context by the DI container at run-time following their class-level directives specifying the scope belonging and defined by the developer at coding-time.

Visibility contexts can be nested giving rise to a hierarchical structure where a context only contains or it is contained by another context (subset property) and a context can not be contained by two non-nested contexts (non-overlapping property). Since the creation and destruction of visibility contexts are triggered by inputs from the presentation layer, the contexts' structure is dynamic and constantly evolving enabling the implementation of advanced application logic mechanisms.

With this organization, the DI container can identify the injection candidates by selecting the instances of the required type by simply selecting those that are within the visibility range of the client instance, if there are no candidates, the DI container will instantiate an instance of the required type and it will associate it to the proper visibility context before the injection in the client component.

The visibility relation introduces a criterion to identify the injectable instances for a given client instance, however slightly different behavior is present among the available frameworks: some completely disambiguate the operation identifying a unique candidate (e.g., CDI and Spring DI are notable examples), others instead offer a more flexible yet error-prone selection through specification setting (this is the case of Angular for instance).

Another aspect that DI containers take care of, is the destruction of the instance components, this feature can be considered as a side-effect of the visibility contexts implementation: as stated above, a context is created and destroyed at runtime following the inputs from the presentation logic, when a context is destroyed, by definition all the component instances associated with the same context become unreachable and since they can not be injected anymore they are destroyed. Considering that the DI container is responsible for the instantiation, the injection and subsequently of the destruction of the dependencies, the dependency injection frameworks offer far more than a simple instance resolution service implementing, in addition, a sophisticated *automated life cycle management* mechanism.

2.4 Dynamic Behavior of the Application Logic

It is clear now that, the application logic represents a crucial aspect of the whole application, especially for the implementation of the functional requirements: it is the junction point between the user interface and the persistence mechanism and as such, it supports the use cases scenarios executions reacting to inputs sent by the presentation layer, maintaining the session state and interfacing with the persistence layer if necessary.

Its role is made even more central with the support of dependency injection frameworks which encourages the implementation of advanced and complex mechanisms additionally, the automated life cycle management feature, enables the components instances living in the application logic to vary among time and user interactions. With this configuration, DI and automated life cycle management orchestrate the execution of multiple concurrent contexts and component instances in a way determined by: the static context assigned to required components at coding time; and by the sequence of users' interactions received at runtime. In this concurrent execution, component instances belonging to different contexts (and in turn different life cycles) interact with each other through method invocations that result in implicit data flow coupling.

To make this coupling explicit is proposed here an abstraction able to visually represent the progressive evolution among time and usage of the application logic, as an example of this, an illustrative instance is depicted in Fig. 2.4.

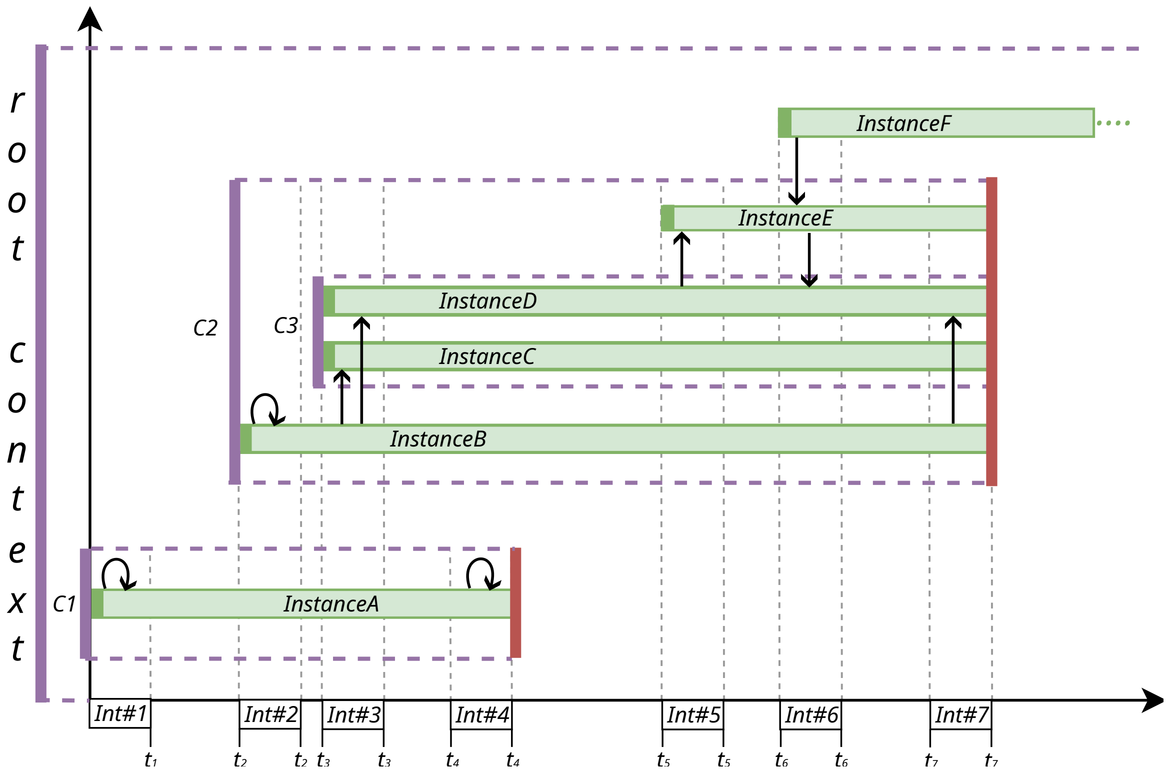


Figure 2.4: Graphical representation of the application logic evolution during a use case scenario execution.

Time is represented continuously, however, the horizontal axis also shows, with irregular frequency, intervals marked with an id in the form *int#n*: these epochs represent the request/response processes triggered by user interactions, and they are modeled as instantaneous intervals (note that the time before and after the interaction is the same) since the time required to generate a response by the application can be considered negligible in comparison with the time spent by a real user between two subsequent inputs.

The abstraction also shows areas with dotted margins that develop horizontally representing the visibility context outlined in Sect. 2.3.1. It is shown neatly that they are created (violet bars) and destroyed (red bars) at different times and that they can contain entirely another context giving rise to the typical nested structures. Note additionally that, when a context is destroyed, all its embedded contexts are destroyed with him.

Inside contexts, associated component instances are represented, each of them lives among multiple requests and can entertain interactions with other instances (represented in the abstraction through solid arrows between the two interlocutors). In this regard, it is crucial to highlight the passive nature of the application logic where interactions between component instances are triggered only within the explicitly demarcated request/response procedures. However, once triggered the application logic, multiple interactions could take place during the answering procedure, to preserve the ordering, interaction epochs are represented as thick bars describing qualitatively the time even though epochs are considered instantaneous. As an example, at context $c2$ opening, instance $InstanceB$ is created inside it, more in detail: at time t_3 the user performs int#3 that triggers a response process in the application logic involving the interactions of $InstanceB$ first with $InstanceC$ and then with $InstanceD$ both belonging to the newly opened context c_3.

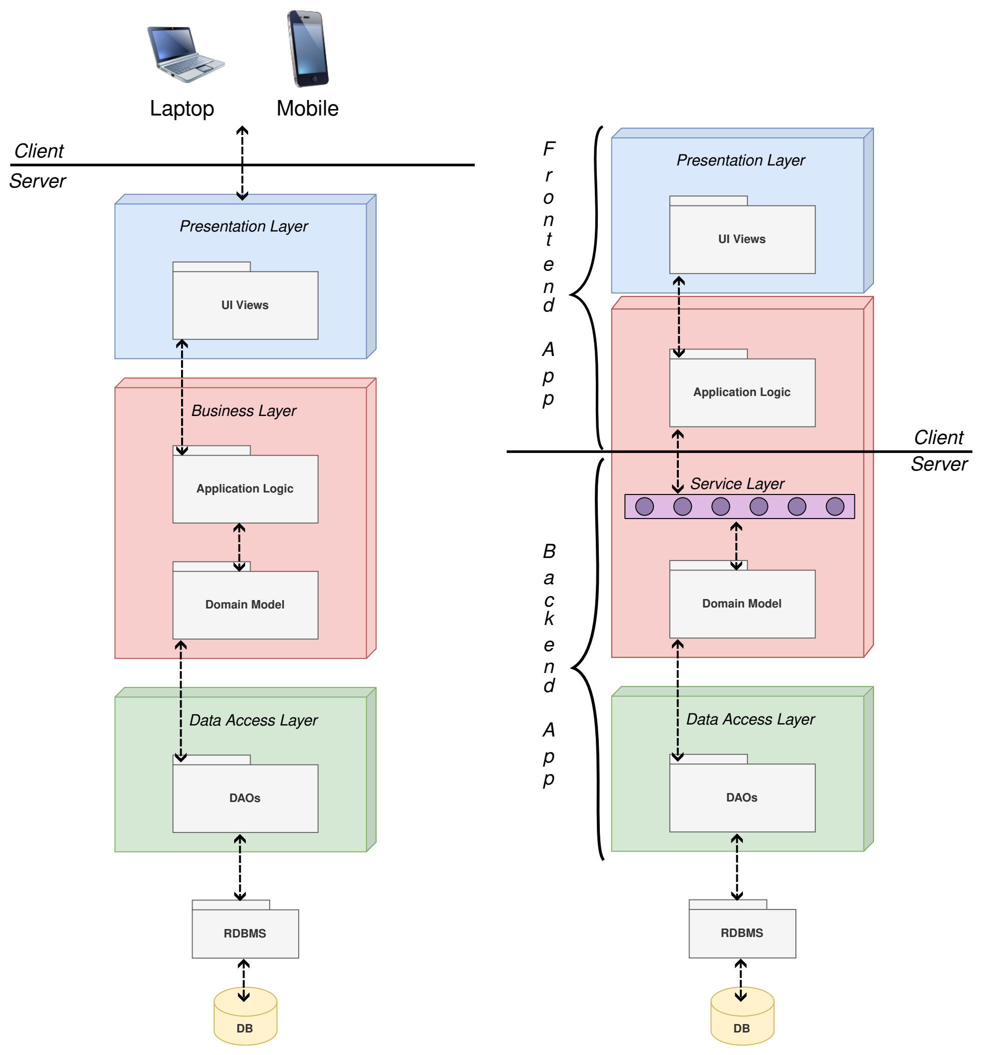
The abstraction also outlines that component instances' life cycles are strictly tied with their associated context and they are destroyed with its closure however, they are not instantiated at the opening of the context but only when they are required (*Laziness principle*), this is the case of $InstanceE$ which is created at time t_5 although its context c_2 was created at time t_2.

Finally, note that the abstraction offers an explicit representation of the reachability range which, for two instances, is bounded by the structure of their associated contexts e.g., again $instanceB$ can interact with $instanceC$ during interaction int#3 because $InstanceB$ is associated with context $c2$ which embeds the context $c3$ where $instanceC$ lives, at the same time $InstanceB$ can not reach $InstanceA$ since their contexts are not nested.

2.5 Stateful VS Service-Oriented Architecture

Nowadays, the application logic is so important that choosing where to run it, and in turn also where to maintain the session state, determines the architectural style to implement.

Assuming a client-server organization of the architecture, which implies a physical separation between the client sending requests, usually corresponding to a desktop or a mobile device, and the server responding to the received requests, the application logic could be implemented on both sides. The advantages and disadvantages involved in this choice are out of the scope of this dissertation however, it is important to point out that implementing a server-side or client-side application logic has also implications on the type of technologies to be used and the way the architecture is organized resulting in two different architectural styles called: *Stateful Architecture*, referring to the fact that the server has a state, or *Service-Oriented Architecture* (SOA) otherwise. Concretely, while a stateful architecture allows implementing a monolithic server-side architecture avoiding burdening the client with computation, a service-oriented architecture relieves the workload of the server but requires the implementation of two distinct applications: the front-end application hosted on



(a) Stateful Architecture (b) Service Oriented Architecture

Figure 2.5: Comparison Between Stateful and Service Oriented Architectures

the client side and the back-end application hosted on the server, see Fig. 2.5 for a graphical comparison.

More in detail, Fig. 2.5 represents two possible implementations of the 3-Layers Architecture, as can be seen, the lower part of the architecture is the same for both implementations: the data access layer is populated with objects implementing the *Data Access Objects* pattern (DAOs), one for each main Entity, which exploit services of an Object Relational Mapping (ORM) framework to perform CRUD operations on the relational tables with guarantee compliance between the instances specified above by the Domain Model and data schemes defined within underlying Relational Database Management Systems.

The upper part of the architecture instead, has important differences in its organization: as can be seen if Fig. 2.5a, in the stateful architecture, above the domain model, the application logic and the presentation layer are hosted on the server meaning that the client sends the requests remaining stateless and that the server is responsible to process the received requests while also maintaining the session state of the ongoing business transactions. In addition, in the case of an application with a GUI, is up to the server to build, populate and then transfer the view to the requesting client.

Conversely, in the service-oriented architecture (Fig. 2.5b) the presentation, the application logic and the management of the business transaction are borne by the client which is now stateful. This more balanced separation identifies then two distinct applications: the front-end application, hosted client-side, and the back-end application, hosted in the server, which, with this configuration, remains stateless and interacts with the front-end application by exposing a set of services that act as *thin-facades* (Fowler (2012)) over the domain model providing simple CRUD-based functionalities.

In the following, it is outlined how everything described so far about the application logic organization, is realized in the field of Web Applications both under the stateful and service-oriented architectural style. In Sect. 2.5.1 is depicted a detailed formalization of how the business logic can be organized server-side and how DI frameworks usually work. In Sect. 2.5.2, with the aim of showing the similarities between server-side and client-side application logic management, a similar although the less rigorous and thorough description for the front-end applications is provided.

2.5.1 Server-Side Application Logic Management, the Web Application Case

In the architecture of *stateful* Web Applications, the server-side commonly exposes an *application logic* that features methods supporting the presentation layer to display pages, accept user interaction, and determine page navigation transitions. The business logic maintains a state of the user interaction spanning along multiple (HTTP) requests in a layer of *Page Controllers* Fowler (2012) and other stateful components (sometimes termed Beans). As a consequence, the state of the application during a user session is split across the participants of a *Model-View-Controller* Deacon (2009): *i*) the running objects representing domain entities (*Model*) Fowler (2012); *ii*) the HTML page currently displayed to the user (*View*); and *iii*) the hierarchy of managed components in the application logic (*Controller*).

Management of lifetime and dependencies among application logic components are usually managed by a Container, according to annotations extending the plain code definition of classes with meta-information. To implement the resolution, the container maintains a runtime representation based on the concepts of scope and context: a *scope* defines a type of policy that the container can enforce in the lifetime and visibility management of required components; besides, a *context* maintains a collection of references to running objects, often termed *contextual instances*, managed under a common scope. During the runtime, the container maintains a set of contexts, and each managed object is associated with a scope specified by the object type.

By nature, scopes in web applications are shaped by concepts of the underlying HTTP protocol and its state management mechanism Barth (2011): components with *request* scope are allocated and maintained on the server-side only for the time between a user request and the server response; besides, components with *session* scope maintain their state along multiple HTTP requests, spanning from the initial contact (or login) to when the application is left (maybe with a logout). In many interaction scenarios, the implementation of a use case requires data to be maintained along a time span shorter than a session but longer than a single request. This is commonly supported by a scope, termed here *enclosed*, whose boundaries are programmatically demarcated by explicit begin/end operations. In the opposite direction, the *application* scope encompasses multiple sessions by multiple users, along with any long-term run from an application startup to shutdown. Finally, most DI containers also support a kind of pseudo-scope that guarantees that a required component assumes the scope of the dependent component where it is injected. In this work, this will be called *conforming*, in contrast with all the other mentioned scopes which will be termed *absolute*. Note that, consistent with what was said in Sect. 2.3.1, this system of scopes forms a hierarchy: a *request* context is always contained in a *session* and possibly in an *enclosed* context; an *enclosed* context is always wrapped in a single *session* context; and the *application* context wraps all the *session* contexts. Since managed components have a lifecycle, frameworks usually provide the possibility to define *post-construct* and *pre-destroy* actions for each component triggered, respectively, immediately after the creation and immediately before the destruction of the contextual instance. Table 2.1 enlists types of contexts supported by some major frameworks for server-side DI and automated lifecycle management in web applications.

Session data are stored and accessed 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 web tokens. Note that multiple contexts and components of the same type may live at the same time, either for the existence of multiple enclosed contexts or for the concurrent usage of the application by multiple users. The selection of the component to be injected and referred is performed by the container, which identifies which are the *visible* instances within the current

Language	Framework	Built-in Context				
		request	*enclosed*	*session*	*application*	*conforming*
C#	*Autofac*	✓	✓	✓	✓	✓
	Spring.NET DI	✓		✓	✓	✓
Java	*CDI*	✓	✓	✓	✓	✓
	Spring DI	✓		✓	✓	✓
	Guice	✓		✓	✓	✓
Python	*Dependency Injector*					✓
	Pinject				✓	✓
	Injector	✓	✓	✓	✓	✓

Table 2.1: Comparison among built-in contexts for main DI frameworks in high-level programming languages C#, Java, and Python.

request.

2.5.2 Client-Side Application Logic Management, the Web Application Case

In the engineering of service-oriented web applications, the application logic is managed client-side by a stateful front-end application, an application usually launched through a web browser, that maintains the session state, implements the navigation logic and consumes the services exposed by the back-end application (Fig. 2.5b).

Client-side application logic has a lot in common with its server-side counterpart: it is populated by stateful components managed by a DI container and it is organized with the presentation layer in a Model-View-Controller scheme. However, there is a substantial difference that comes into play switching from a server-side to client-side application logic consisting in how the components' life cycles are structured: in a stateful architecture, components maintain their state among multiple HTTP requests and their scopes are shaped by the concepts of HTTP, conversely, in a service-oriented architecture, the front-end application loses its tight connection with the protocol since it relies on the HTTP only to communicate with the back-end application invoking the exposed services. In this case, then, the application logic is strictly related to the user interface and the user interactions assume now a central role.

Modern user interfaces are designed with a compositional structure, inspired by the composite pattern Gamma et al. (1995), where graphical elements are defined through the definition of their sub-elements and so, concretely, the development of a front-end application is usually based on the so-called *Component Driven Development* Frost (2016); Godbolt (2016) where a single element of the user interface, the *widget*, can be developed in isolation through the specification of its template - which defines both its composition with other widgets and its appearance - and the specification of its controller - which defines the logic underneath.

This strategy enforces the coupling between the presentation layer and the application logic and identifies a finer-grained version of the page controller pattern where a controller drives a single widget rather than an entire page.

For the same reasons described in Sect. 2.2, along with controller components, it is also possible to define stateful helper components detached from a specific graphical element that can be injected and shared among other components through class-level specifications.

In this configuration, controllers live as long as the widget is visualized in the user interface and the helper components can be injected and shared at different levels of the widget composition also inheriting the same visibility range and life cycle. This delineates a similar scenario to that outlined for the stateful architectures where here the visibility context features are implemented by the concept of widget. A context can be characterized by nested structures and if it is the case, it ties its destruction to the destruction of its sub-widgets, additionally, it also defines a scope and a life cycle to be associated with: the related controller and the injected helper components have a determined visibility and a life cycle that span multiple user interactions.

This delineates a scenario equivalent to that outlined in Sect. 2.4 and in turn similar to the case of the server-side application logic shown for the stateful architecture case. In particular, in this case, the concept of widget acts as a visibility context for the dependency injection container:

- it allows the definition of nested composite structures where the destruction of a widget determines the destruction of all its embedded widgets;

- it defines a visibility scope: a component has a determined visibility that depends on the structure to which the associated widget belongs;

- it defines a life cycle binding its destruction to the destruction of its associated components.

It is worth noting that, DI frameworks for client-side application logic do not provide built-in HTTP-based scopes since as anticipated above, the HTTP ceases to play a central role in the application logic organization and session state management, this paradigm shift impacts heavily on the contexts structure form, which now allows an unbounded nesting level, and also on the component instances' life cycles that now span among multiple user interactions instead of requests.

2.6 Software Architectures, from Theory to Practice

In this section, concepts are illustrated in a concrete setting through the discussion of two real architectures implementations: *FlightManager* (Sect. 2.6.1), which is a full-fledged mid-sized stateful web application implemented with the Java/Jakarta Enterprise Edition (JEE) technological stack, and *ToDoApp* (Sect. 2.6.2), a simple front-end application developed in Angular.

2.6.1 FlightManager, a Stateful Architecture Case Study

It is introduced here an exemplary application, named *Flight Manager*[1], developed according to a common artifact-driven good practice, implementing widespread architectural patterns common to a large class of *stateful* Web Applications, *Flight Manager* is a mid-size stateful web application implementing a sound and widely adopted combination of architectural patterns, developed at the *Software Technologies Lab* of the University of Florence by software professionals with strong and consolidated experience, with the aim of supporting experimentation in research and providing a reference implementation for teaching in a master course on software architecture and methodology. Functions and architecture are described here for the purposes of the present treatment, and more extensively documented in the accompanying repository.

Flight Manager features functions in the context of an online flight booking system, selected without loss of generality as a context that can be intuitively understood without specific domain knowledge, represented in the Use Case Diagram in Fig 2.6: main actors are application administrators and customers that may want to book a flight both as occasional users and as frequent users with a personal account; a Visitor can search flights connecting airports and then book and manage reservations for multiple passengers; in addition, a user in the role of the Registered customer, can obtain special discount rates.

The application structure implements a 3-tier *stateful* architecture, outlined in the *UML Deployment Diagram* of Fig. 2.7: the Domain Model Fowler (2012) is composed by 10 classes (partially) represented in the Class Diagram depicted in the *Domain Model* package of the figure; the Data Source is implemented by a Relational Database Management System (RDBMS), connected to the Domain Model by a Data Access Layer, featuring a Data Access Object (DAO) for each main Entity (for a total of 6 DAO classes), which exploits services of an Object Relational Mapping (ORM) framework; the Presentation Layer is organized in a User Interface made of XHTML pages (roughly, 30 pages, shown in Fig. 2.8) and a Business Logic Layer that features

[1]made publicly available for the research community at `https://github.com/LeonardoScommegna/unravel-experimentation`

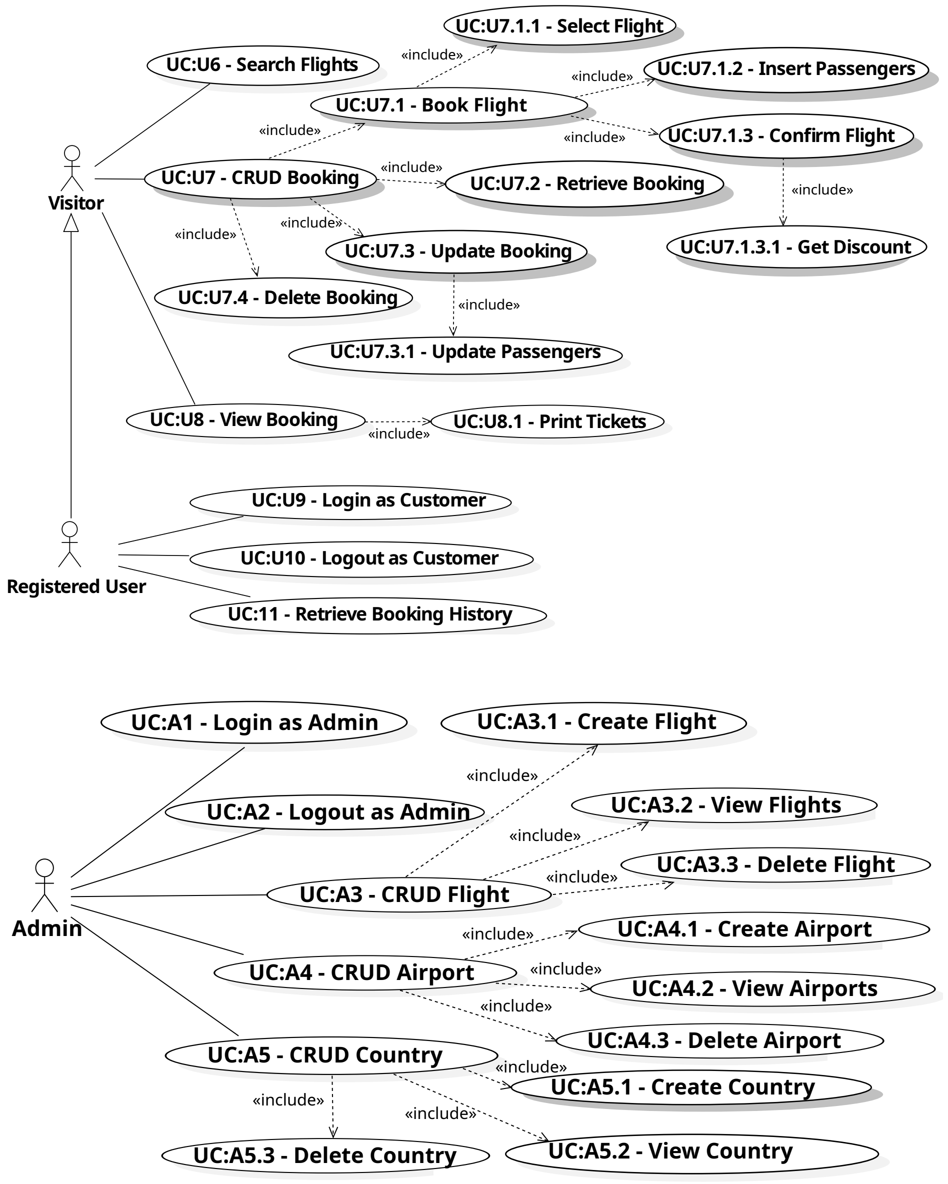


Figure 2.6: Use case diagrams of *Flight Manager*.

Controllers implementing the navigation logic and other components maintaining information accumulated along the user interaction (roughly, 30 classes).

Without loss of generality, the architecture of *Flight Manager* is implemented by adopting specifications of the Java/Jakarta Enterprise Edition (JEE) ecosystem. In particular, ORM relies on the *Java Persistence API* (JPA, here provided by Hibernate), and interface XHTML pages are based on *JavaServer Faces* (JSF). More importantly for the present treatment, DI and automated lifecycle management for objects running in the Business Logic layer is implemented using *Contexts and Dependency Injection* (CDI, here provided by the Weld reference implementation). Comparable specifications and technologies are available in other ecosystems for Web Applica-

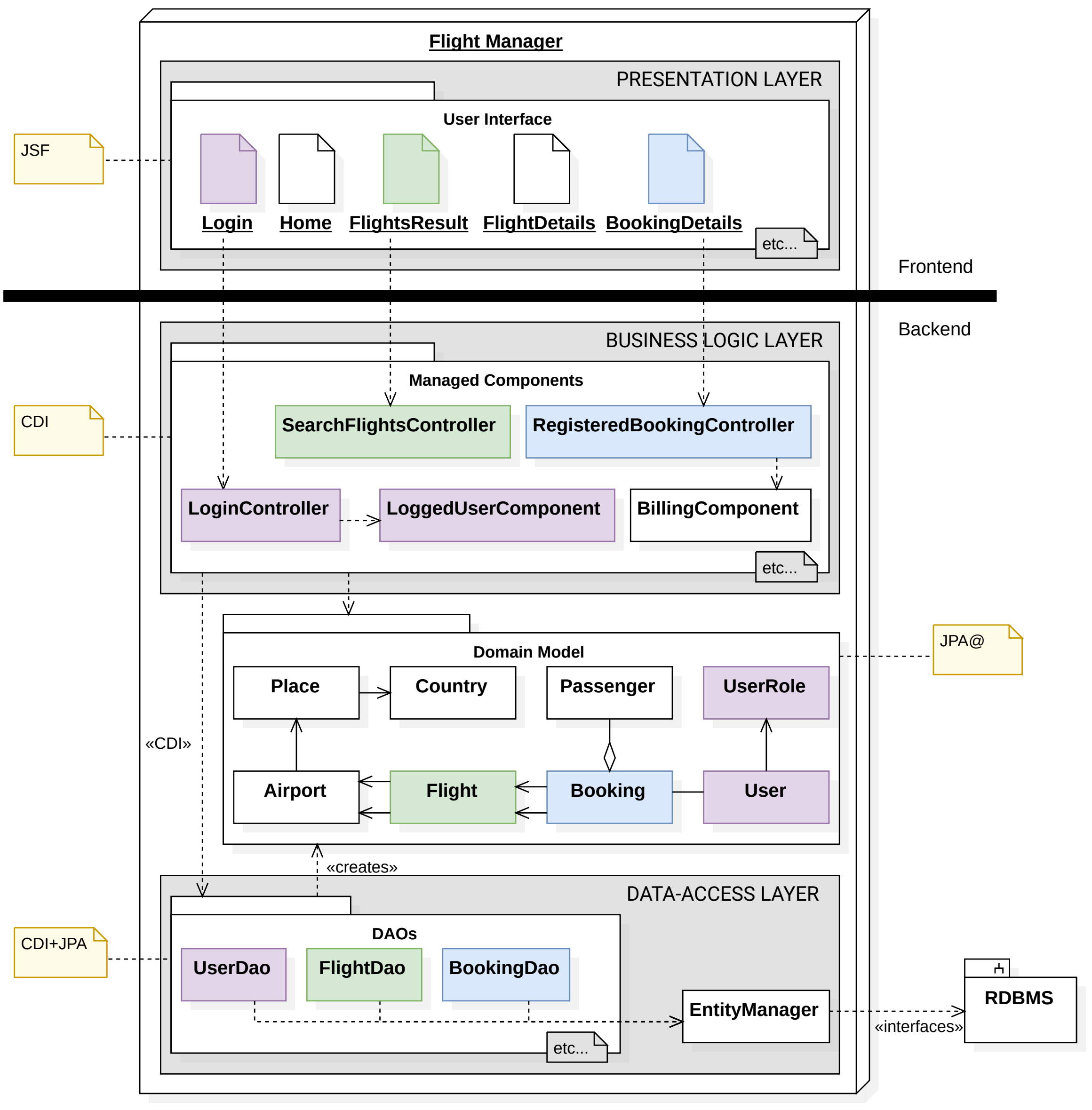


Figure 2.7: Architecture of *Flight Manager*.

tion development. In particular, functionalities and concepts of CDI are paralleled by Spring-DI in Java, Autofac in C#, and other frameworks outlined in Table 2.1.

The CDI framework of JEE provides tailored annotations for specifying that a Java class be managed as a bean associated with one of the *built-in* contexts *@RequestScoped* (i.e., request), *@SessionScoped* (i.e., session), *@ApplicationScoped* (i.e., application), *@ConversationScoped* (i.e., enclosed) and, *@Dependent* (i.e., conforming). On the other hand, the annotation *@Inject* specifies that the target of a reference variable is managed by the container as a required component. The snippet in Fig. 2.9-left illustrates the concept showing the definition of class *LoginController*, which can be injected as a component with request scope (specified by the annotation *@Re-*

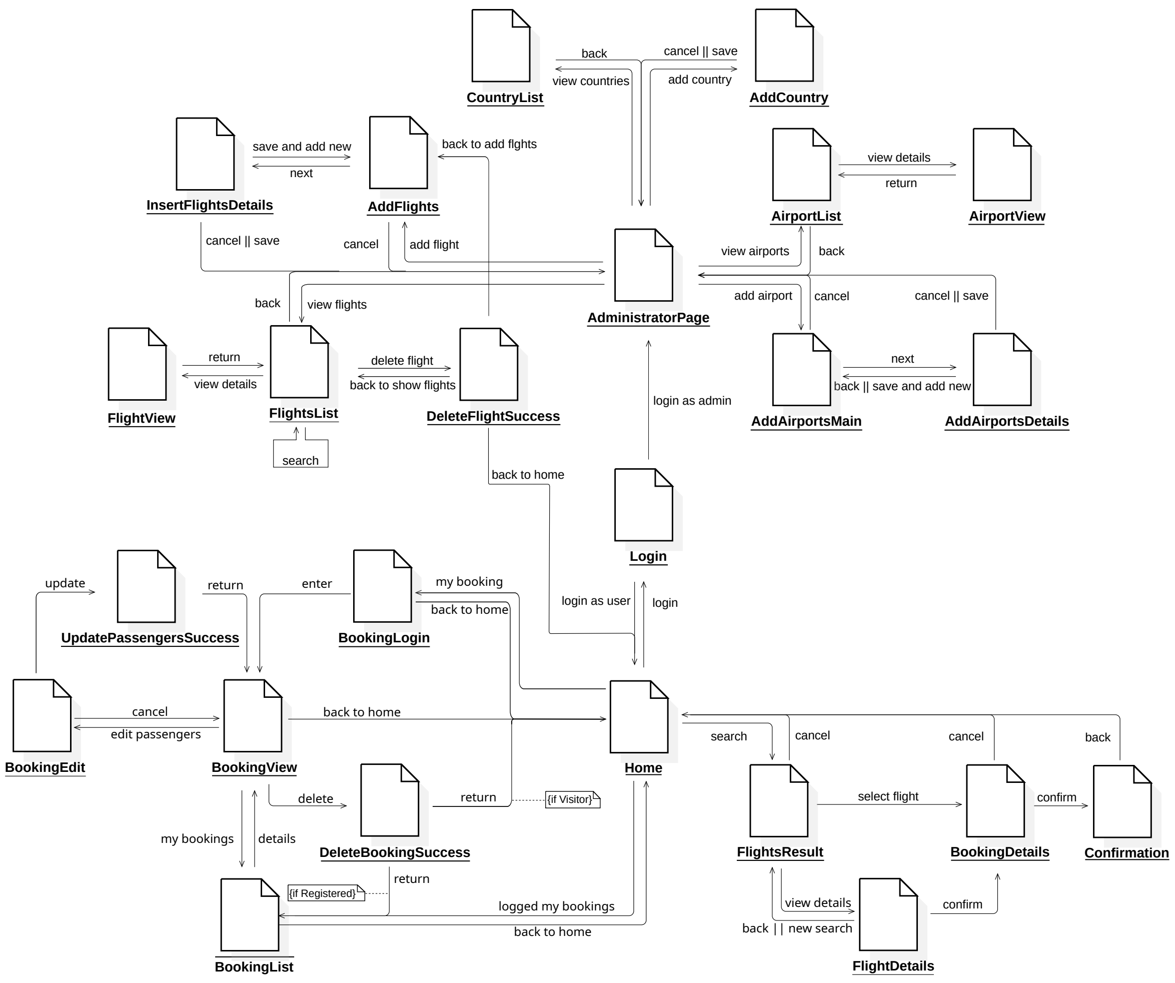


Figure 2.8: Page Navigation Diagram of *Flight Manager*.

questScoped stacked on the class definition), and which in turn receives an injection of three components in the types *UserDao*, *LoggedUserComponent*, and *PasswordManagerComponent* (specified by the annotations *@Inject* stacked on the declaration of reference variables that shall point them).

Note that the scope is a static property of each type of injectable component, specified by annotation of its class definition, and it is neither controlled nor made explicit at the injection point where the component is required: in particular, the scope of instances referred by *userDao*, *loggedUser*, and *pwdManager* are not known to the code of *LoginController*, and they shall rather be retrieved by inspection of the annotations of the separate code of their class definitions. This practice, which is common to DI and automated lifecycle management in all major frameworks, largely limits design control and results in error-prone programming approaches, exacerbated further when components are developed and reused by separate teams with different practices, knowledge, and skill Sharma and Spinellis (2018). To cope with this difficulty, Fig. 2.9-right shows a fragment of a *UML Class Diagram* with (informal)

```java
@RequestScoped
public class LoginController {
  private String username;
  private String pwd;
  private String toHome = "index";

  @Inject
  private UserDao userDao;
  @Inject
  private LoggedUserComponent loggedUser;
  @Inject
  private PwdManagerComponent pwdManager;

  public String loginAsCustomer() {
    User u = userDao.login(this.username,
        pwdManager.encode(this.pwd));
      if(u != null)
        this.loggedUser.initUser(u);
      return (u == null) ? "" : this.toHome;
    }

  public String logout() {
    this.loggedUser.shutDownUser();
    return this.toHome;
  }
}
```

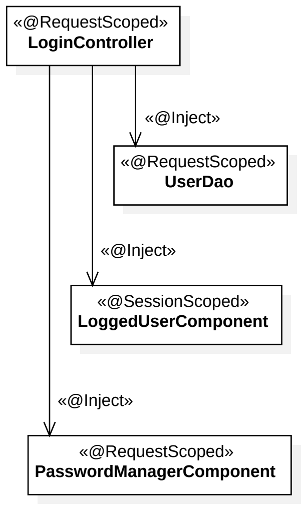


Figure 2.9: (*left*) definition of the Java class *LoginController*, which handles the *Flight Manager* login page. Username and password fields are supporting variables used by XHTML forms in the Presentation Layer; *userDao* and *pwdManager* are injected components, representing dependency relationships, and *loggedUser* is initialized after successful authentication; the *loginAsCustomer*() method uses both dependencies to apply the right database query. In case of authentication, the end-user is redirected to the *Home* page (the *toHome* attribute is initialized with a return string written in the JSF syntax);
(*right*) UML Class Diagram annotated so as to identify injected dependencies and components scopes: *LoginController*, *UserDao*, and *PasswordManagerComponent* (in *request* context); *LoggedUserComponent* (in *session* context).

stereotypes specifying the scope of injected components and identifying dependencies that will be managed by the DI container.

A possible basic development methodology tailored for the characteristics of the architecture of Fig. 2.7 is illustrated in Fig. 2.10 as a Data Flow Diagram. Displayed pages are initially identified from Use Case Diagrams and Templates Curcio et al. (2018), sketched in mockups and organized in a *Page Navigation Diagram*, shown in Fig. 2.8 for the *Flight Manager* application example. Following the ICONIX process,

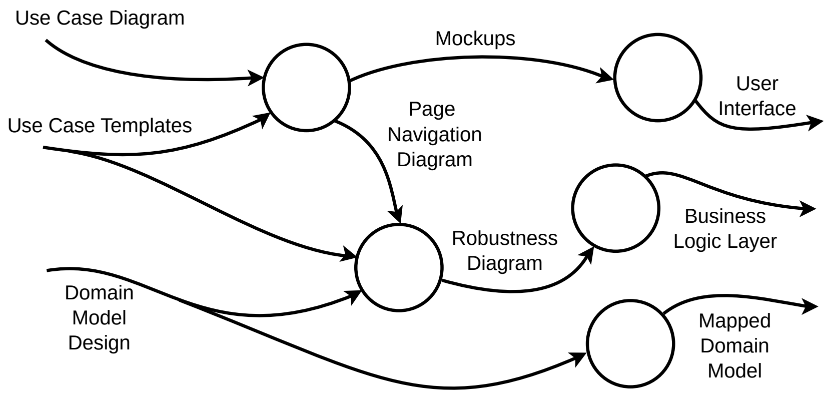


Figure 2.10: A basic artifact-driven development process for a Web Application in the *Flight Manager* architecture.

pages can then be refined through Robustness Analysis to associate user interaction on displayed pages (represented as *Boundaries*) with objects in the underlying business logic and domain model (represented as *Entities*), so as to obtain a Robustness Diagram for each Use Case Rosenberg et al. (2005). For example, Fig. 2.11 illustrates the *UML Robustness Diagram* for the "View Airports" use case (UC:A4.2 in Fig. 2.6). Note that the diagram represents the navigability between *Boundary* elements, in light blue, (i.e., pages *AdministratorPage*, *AirportList*, and *AirportView*) and navigation actions (e.g., *nav back* or *nav details*). The diagram also reveals what happens at the traversal of a navigation edge, through the *controller* elements, in light green, and the access to underlying *Entity* elements in light orange (e.g., from page *AirportList*, some details of an *airport* are retrieved during the navigation action *nav details*).

2.6.2 ToDoApp, a Front-End Sample Application

It is presented here *ToDoApp*, a simple front-end application that supports the task management of a user. As can be seen by the use case diagrams in Fig. 2.12, *ToDOApp* allows one to create a project and then populate it with various tasks. Each task has a title and a field that marks if it is already done or not, optionally, a task can be decorated also with a priority level ranging from 1 to 10 and enriched with a textual description.

The application is entirely developed in Angular 2 exploiting the built-in Dependency Injection framework for the management of the application logic. It is worth mentioning that, since it is only a front-end application, it is not a complete software architecture lacking server-side layers. This results in a lack of persistence of the application meaning that all the tasks added during the usage will be deleted once the application is closed. However, the persistence feature could easily be added with minor modifications to *ToDoAPP* simply including calls to services exposed by an

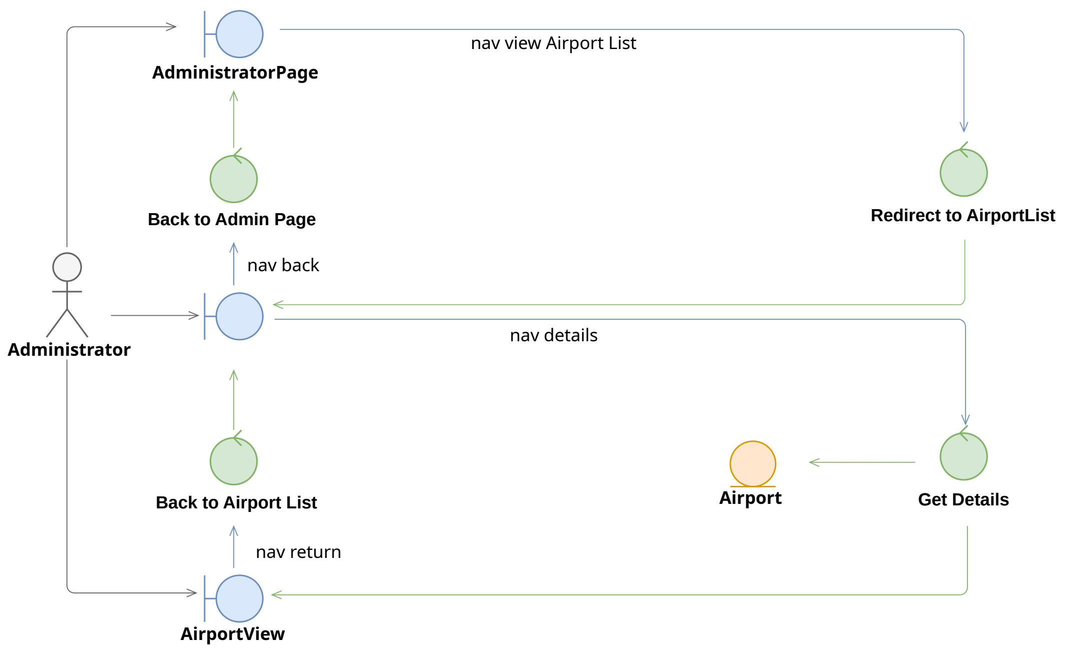


Figure 2.11: UML Robustness Diagram of “View Airports” admin use case (i.e., UC:A4.2): starting from *AdministratorPage*, by clicking *nav view Airport List* the admin is redirected to the *AirportList* page. At any time, by clicking *nav back*, a forward to the *AdministratorPage* page is performed while *nav details* and *nav return* actions allow to inspect information about a specific *airport* in the *AirportView* page and then come back to the main list. In the diagram, *entities* represent domain model objects as ○̲ (orange), *boundaries* represent web pages as |○ (blue), and *controllers*, represent managed components as ↻ (green).

ad-hoc back-end application.

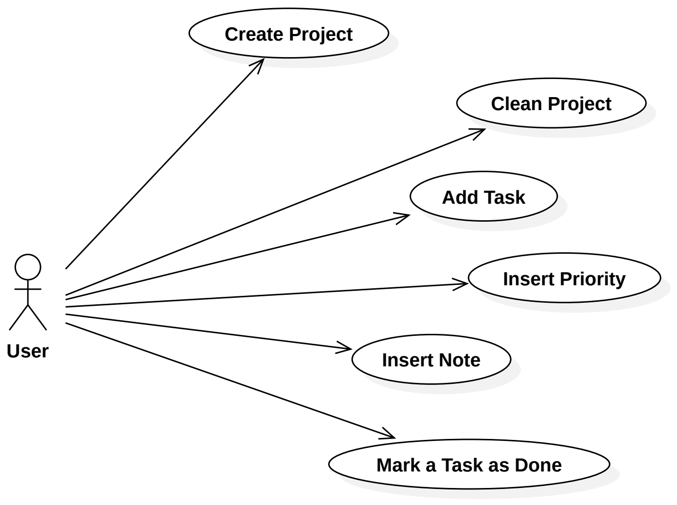


Figure 2.12: Use case Diagram of *ToDoApp*.

Chapter 3

Characterizing Threats and Testing Application Logic

Productive development of Application Logic in Software Architectures largely resorts to Dependency Injection and automated management of life cycle and visibility of injected components. While promoting separation of concerns in design and implementation, this practice hides runtime usage relationships and concurrency among software components, blurring the overall structure and state of the application and thus hurdling safe control along the development and verification stages.

This chapter characterizes the chain of threats induced by the use of DI and automated life cycle management in application logic with stateful components, identifying the specific fault model that they subtend and the failure modes that might result additionally it is proposed a semi-automated Model-Based Testing (MBT) approach Utting et al. (2012) that counteracts it.

To this end, an abstraction is proposed, named *managed components Data Flow Graph* (mcDFG), which makes explicit data flow coupling occurring among components under the orchestration of the container in the execution of each application use case. It is then shown how this abstraction can be derived by the automated transformation of annotation of basic artifacts of a disciplined SW development life cycle, and how it can be used to select test cases according to various criteria of data flow coverage Rapps and Weyuker (1985). The proposed approach is illustrated and experimented on a suite of mutations of an exemplary web application, named *Flight Manager*, which implements widespread architectural patterns of the good practice of software engineering.

In this Chapter, motivation and related works are drawn in Sect. 3.1 and Sect. 3.2 respectively, in Sect. 3.3 the fault to failure chain related to the application logic management is depicted identifying first a group of failure modes, and then the associated fault model; the proposed testing methodology is shown in Sect. 3.4 and experimentation with related results are present in Sect. 3.5.

3.1 Motivation

DI and automated lifecycle management provide essential functionality for the productive development of enterprise-scale applications, promoting abstraction, separation of concerns, and reuse, and it becomes definitely crucial in the application logic where the state of user interaction is maintained in a *business logic* layer Manuel and AlGhamdi (2003) made of *Page Controllers* Fowler (2012) and other stateful components (sometimes termed Beans).

However, DI and automated lifecycle management also reduce designer control over the intertwined effects of *concurrency* orchestrated by the container and *dataflow coupling* among the instances of managed components that overlap their activity. In fact, actual dependencies among instances are determined during the runtime as a joint effect of the sequence of inputs issued by users' navigation of application pages and structural characteristics of application logic components, according to mechanisms that remain implicit and often only partially understood by SW developers.

This gives raise to a specific chain of threats Avižienis et al. (2004), with types of faults related to how components are associated with a lifecycle model and how they are composed, and with subtle mechanisms of fault-to-failure propagation that are often hard to activate and observe.

Maintaining reliability while exploiting the potential of DI and automated lifecycle management requires suitable means to provide a view of the effects of design choices and to verify an implementation through effective tests focused on specific types of fault and interpreted by an oracle able to identify error states that may be not observable at the user interface.

3.2 Related Works

In the literature of software analysis, design, and testing, software systems, and in particular Web Applications, have been modeled under various perspectives Diehl (2007). Abstractions capturing static and *structural* characteristics of a system enable fine modeling of software components in isolation or in mutual dependence, relying on the knowledge extracted from design documentation or implementation artifacts. Besides, dynamic *navigational* and *behavioral* abstractions provide capabilities for reducing the complexity of the verification problem by representing only feasible sequences of operations and system behaviors in accordance also with functional requirements specifications and use cases.

Structural characteristics in Object-Oriented (OO) applications are commonly represented using the *Unified Modeling Language* (UML) Booch (2005) and its available *UML profiles* Fuentes-Fernández and Vallecillo-Moreno (2004). In Conallen (1999, 2003), the *Web Application Extension* UML profile is presented for supporting design activities for Web Applications through the abstraction of *ad hoc* primitives about pages, forms, links, redirects, scripts, and style sheets. While in Souza et al. (2007), the *FrameWeb* UML profile is proposed for supporting designers in modeling web information systems based on specific types of frameworks (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).

Further abstractions, inspired by *Control Flow Graph* (CFG) Allen (1970) and *Data Flow Graph* (DFG) Rapps and Weyuker (1985), provide graph models focusing on the OO paradigm Ferrante et al. (1987); Souter et al. (1999) for representing dependency relationships among dynamic and reusable components, also integrated within distributed systems and orientated towards web development. In Wu et al. (2000) the *Component Interaction Graph* (CIG) is proposed to enable representation of collaborative relationships and dependencies among software components providing a structural overview of modeled interactions by depicting components interfaces as nodes, and dependencies as edges, conceptually identifiable as events (e.g., user actions or interface invocations). In order to support reliability analysis processes over component-based applications, a probabilistic model adapted from the CFG principles is proposed in Yacoub et al. (2004) for the identification of architectural dependencies among components. This abstraction, named *Component-Dependency Graph* (CDG), is a directed graph whose nodes are components, and whose directed edges are transitions between components, each one in turn annotated with details about its estimated reliability and its execution probability. In Shatnawi et al. (2017) a *Dependency Call Graph* is proposed to represent key aspects for the modernization of Service-Oriented Architectures of monolithic legacy systems. Specifically, dependencies that commonly remain hidden are expressed through a language-independent meta-model termed *Knowledge Discovery Meta-Model* Pérez-Castillo et al. (2011), enlightening dependencies related to containers regulated through Remote Method Invocations.

As a common trait, all these component-based graph abstractions are not aware of the concurrency scenarios that underlie Web Applications exploiting DI and automated lifecycle management: they lack expressiveness about components scopes, their visibilities, and their lifecycles boundaries, as well as proxy and interceptor entities, automatically acting in the background through a DI container.

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 exacerbates the need for modelling their navigational characteristics; consequently, some semi-formal and formal standards have been introduced.

In the practical experience, functional aspects of Web Applications are expressed through a simple and intuitive abstraction, named *Page Navigation Diagram* (PND) Kung et al. (2000), characterising the navigation design through the definition of a finite state machine where web pages act as states and hyperlinks as transitions. Otherwise, functional aspects can be expressed also through behavioural UML diagrams; above all, a widely adopted artefact is the *UML Robustness Diagram* Rosenberg and Scott (1999), subtending a reachability graph, decorated with dependency relationships among actors, pages and page controllers, thus mixing information derived both from functional and structural perspectives. Internal behaviours can be also captured in a static or dynamic perspective adopting structural models, such as UML Class Diagrams, as addressed by Ricca and Tonella (2001) with the aim of extending the representation to navigation data flows. However, this latter work does not consider the case of implementations exploiting containers for DI and automated lifecycle management.

Model-Based Testing (MBT) Utting et al. (2012); Anand et al. (2013) 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 System Under Test (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 Schmidt (2006), improving the quality of functional requirements, to the automated generation of tests and systematic coverage of test suites Legeard and Utting (2010).

In a *white box* perspective, MBT techniques perform structural testing for verifying the correctness of the SUT, exploiting the source code and implementation details together with modelling abstraction for test case generation and selection. Among structural testing techniques based on graph abstractions, the most relevant are *Control Flow Testing* (CFT) Beizer (2003) and *Data Flow Testing* (DFT) Rapps and Weyuker (1985); Frankl and Weyuker (1988) based on CFG and DFG respectively. Later, various solutions Harrold and Rothermel (1994); Souter and Pollock (2003); Denaro et al. (2008) have been proposed so as to adapt DFT for the case of OO programming, thus covering *def-use* couples at different levels of granularity by modelling also relationships among attributes and methods of different classes. In Liu et al. (2000), 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, as well as in XML and HTML documents.

Conversely, MBT techniques under a *black box* perspective perform functional testing verifying the conformance between a SUT and a specification, neglecting structural aspects of a system in favour of the adoption of functional or navigational design abstractions, such as software requirements or use cases, describing application business scenarios Tiwari and Gupta (2015). *Scenario-oriented* testing practices, also known as interaction-oriented, describe all reasonable runtime interactions between the SUT and the sequences of inputs or outputs from the end-user point of view Vieira et al. (2006); Nebut et al. (2006); Kaplan et al. (2008). Another significant notation category is the *state-oriented* Offutt and Abdurazik (1999); Kuliamin et al. (2003); Bouquet et al. (2007), which describes the SUT by reactions on inputs and outputs through *finite state automata*, laying its foundations on the consideration that the system behaviour 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).

This section proposes an approach able to fill the gap between the white box and the black box perspective in software architecture and in particular in web application testing and that takes advantage of the best of both worlds. On the one hand, the white box perspective puts the focus on a single application module (or a limited group) allowing an in-depth verification of the modules with the drawback of a high number of test cases and the lack of possibility for containers to operate during test execution. On the other hand, the black box perspective allows containers to be running and to identify considerably fewer test cases but lose control of the implementation details and component states. The presented methodology addresses the SUT verification taking into account of DI and automated lifecycle management mechanisms. It is based on an abstraction (that can be considered in a *grey box* perspective) aware of both navigational design and implementation details regarding container configuration: the test case selection phase is then guided by revised coverage criteria derived from DFT approaches. Thus, obtained test cases, result constrained by allowed end-user sequences of inputs and at the same time are built to be able to inspect how the state of the components managed by the container reacts to these stresses.

3.3 The Chain of Threats

In this section, the chain of threats affecting the development of the application logic of software architecture is characterized by classifying types of coding *faults* (Sect. 3.3.1) and *failure modes* that they can produce (Sect. 3.3.2).

Then, in Sect. 3.3.3 it is exemplified how fault types can occur in realistic scenarios and illustrate how they can be activated and propagated by user actions in the navigation of interface pages.

3.3.1 Fault Model

It is here considered a taxonomy of fault types that can be introduced in annotation or programmatic lifecycle specification, which makes the scope of a managed component unfit for the needs of the point where it is injected.

- **ShorterScope:** a component is assigned an absolute scope *lower* than what would be required.
- **LongerScope:** viceversa, a component is assigned an absolute scope *higher* than what would be required.
- **WrongConformance:** a component is assigned a *conforming* scope while it should have been *absolute*, or vice versa.
- **EarlyOrUndueClosure:** the *end* demarcation of an enclosed context is erroneously added or placed too early in the code.
- **LateOrMissingClosure:** the *end* demarcation of an enclosed context is missing or it is placed too late in the code.
- **LateOrMissingBegin:** the *begin* demarcation of an enclosed context is missing or late in the code.
- **MissingStateClearance:** the code misses a required clear-out or re-initialization of a component, which should be triggered at the creation or destruction of some other component as a post-construct or pre-destroy action.
- **ErroneousDynamicInjection:** the type of an injected component is erroneously determined, which may occur when injection types is determined dynamically during the run-time.

This taxonomy reflects structural characteristics of annotation-based or programmatic specification of the lifecycle of managed components, and it covers major complexities observed in a long-termed experience of development of stateful web applications (e.g. an Electronic Health Record in use for several years in a major Hospital of Tuscany Region Patara and Vicario (2014)Fioravanti et al. (2016)). It also covers issues reported by developers with different levels of skill in technical social forums like *StackOverflow*, *Github*, and *DZone* (partially documented in the additional materials in Chapter A), and difficulties and limitations encountered by tens of students developing JEE stateful components in assignments of a master level course active for more than 5 years.

3.3.2 Failure Modes

Faults in annotation and programmatic specification of managed components life cycle may result in various kinds of *errors* in the type of injected components or in the logic of the intervals Allen (1983) during which they exist, maintain their state, and are shared by multiple dependants. In turn, this may cause various types of deviations in the functional behavior delivered by the user interface.

To deepen this aspect, are identified four types of *failures* occurring when an injected component: does not maintain memory as long as required (*vanishing component*); or, vice versa, it is not renewed when needed (*zombie component*); or it becomes visible at the same time to multiple dependants that should not share it (*unexpected shared component*); or it is created in a wrong type variant (*unexpected injected component*).

Vanishing component. An injected component may not live and maintain its state with continuity along the time interval needed by its dependants, thus resulting in a null pointer exception or a data loss (if the component type is restarted by a new injection), as illustrated in Fig. 3.1.

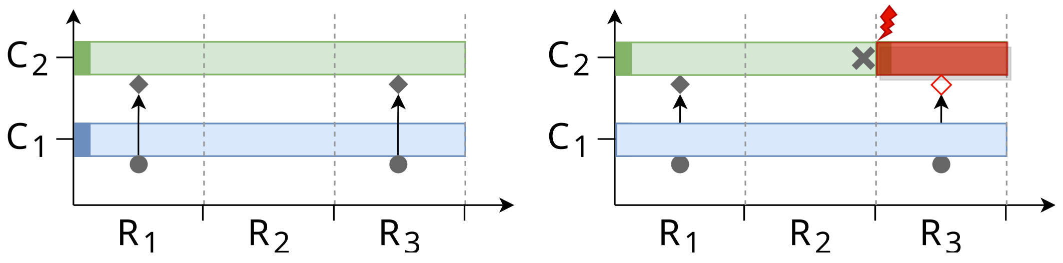


Figure 3.1: *Vanishing component failure.* (left) the expected correct behavior in some scenario with two coupled instances ● and ◆ living in distinct contexts C_1 and C_2: ● uses ◆ twice expecting that this maintains its state across subsequent requests. (right) a faulty behavior: at the beginning of R_3, context C_2 is restarted (instead of continuing) and the DI container constructs a new instance ◇ of the same component type; the fault is activated at the point marked by ⚡, entering an erroneous state that produces a data loss failure when ◇ is used by ●.

Zombie component. In the opposite situation, an injected component may remain alive with continuity while a dependent component expects that it is destroyed and restarted. This may lead to components that maintain an obsoleted state, as illustrated in Fig. 3.2, or it may also potentially produce an aging failure due to memory leakage Grottke et al. (2008).

Unexpected shared component. A context may remain continuously active so as to be accessible by two or more concurrent dependent contexts. This may lead multiple dependants to erroneously share the same instance of some required component,

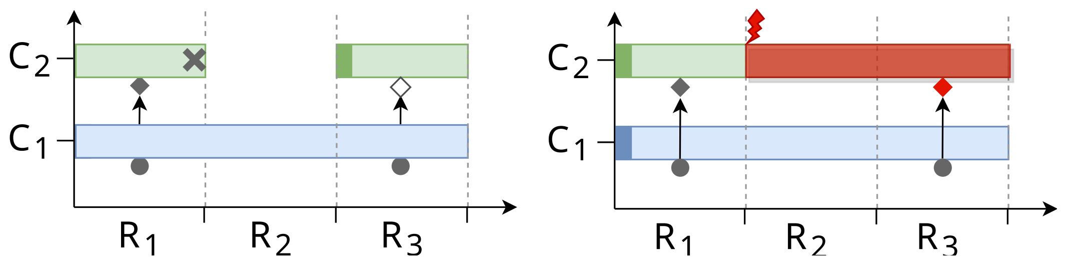


Figure 3.2: *Zombie component fault.*(left) in a correct implementation, ● should access two distinct instances of ◆. (right) however, since the context C_1 is not closed and restarted, the instance ◆ retains memory also during R_2 and the second access of ● will find an obsoleted and not refreshed state.

causing failures due to interference on the component state, with complex race conditions that may behave as Mandel- or Eisen-bugs Grottke and Trivedi (2007), as illustrated in Fig.3.3.

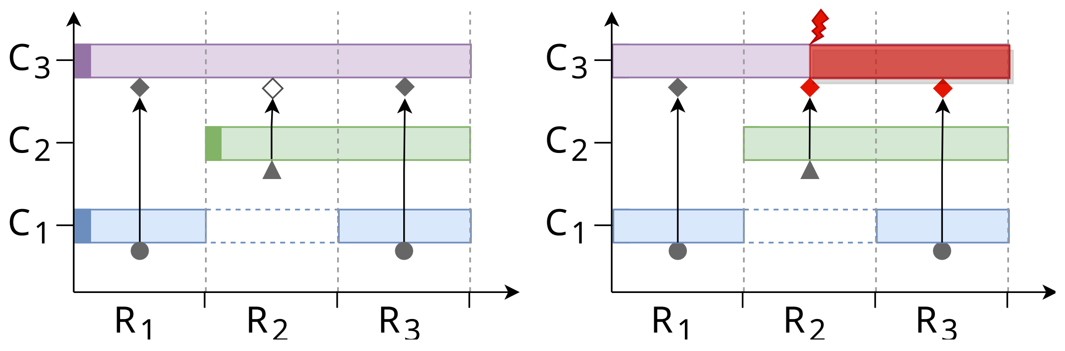


Figure 3.3: *Unexpected shared component fault.* (left) The ● and ▲ contextual instances expect each one to inject a different instance of the required component (i.e., ◆ and ◇, respectively). (right) yet, the DI container resolves both dependencies with the same contextual instance, thus producing interference and unpredictable race conditions.

Unexpected injected component. The type of a required component may be wrongly specified at its injection point, for trivial coding error or for subtle defects in the static selection of alternative implementations of a type or in the logic of a dynamic programmatic lookup. this may cause a variety of deviations from the expected use case flow, unpredictably leading to fast failure or complex aging effects Grottke et al. (2008). Fig. 3.4 illustrates the concept.

Identified fault and failure types have some typical causal relation, which may direct analysis of root causes: *vanishing components* naturally result from ShorterScope, EarlyOrUndueClosure, and LateOrMissingBegin faults; conversely, a *zombie component* can be easily caused by LongerScope, LateOrMissingClosure and MissingState-Clearance faults; an *unexpected shared component* can be produced by the same faults

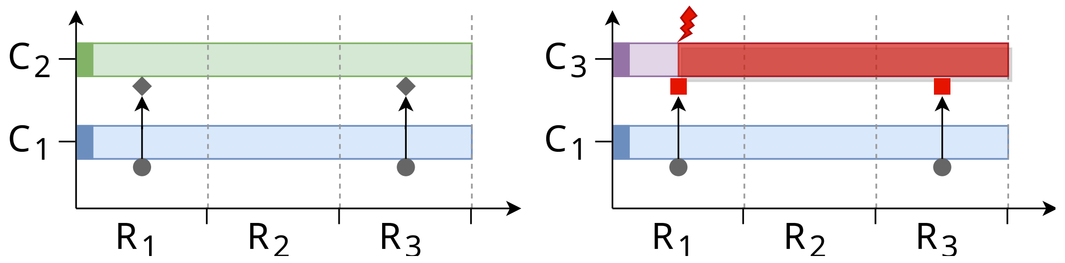


Figure 3.4: *Unexpected injected component fault.* The DI container, in R_1 resolves the dependency of ● with a wrong contextual instance (i.e., ■ instead of ◆), thus producing unpredictable behaviours.

that cause a zombie component but with a different mechanism; all failures due to longer or shorter scope can also be due to a WrongConformance, with effects depending on the specific mismatch between conforming and absolute expected components; finally, *unexpected type* typically results from an ErroneousDynamicInjection.

3.3.3 Failure Logic scenarios

It is illustrated here how each failure mode can be produced by the activation of a fault occurring in the execution of some use cases of the Flight Manager and the ToDoApp examples. More comprehensive coverage of all fault types is reported in the experimentation section (Sect. 3.5). Examples are selected so as to show how faults can be realistically introduced, even in good practice of SW development of both server-side and client-side application logic, and how they can be activated by specific but realistic paths in the navigation of interface pages.

Vanishing components. The occurrence of this kind of failure is exemplified in the "Search Flights" use case (UC:U6 in Fig. 2.6) where the user repeatedly visits *FlightDetails* pages (see the PND in Fig. 2.8). To this end, the state of the *FlightsResult* page must be maintained by an *enclosed* context begun at the initial traversal of *search* from the *Home* page and ended at the traversal of *selectFlight* from *FlightsResult* or *confirm* or *newSearch* from *FlightDetails*. However, various faults can be introduced in the begin/end programmatic demarcation of the *enclosed* context.

If the controller activated at traversal of *back* from *FlightDetails* includes an end of context (*EarlyOrUndueClosure* fault), a *commission failure* McDermid and Pumfrey (1994) occurs in the interaction of the code with the container, and the page *FlightResults* loses memory, causing an occurrence of the *vanishing component* failure.

Besides, if the controller that serves traversal of a *newSearch* does not include an end of context (*LateOrMissingClosure* fault), an *omission failure* Bondavalli and Simoncini (1990) occurs and the page *FlightResults* maintains a stale state memory as in the *zombie component* fault, which can later produce a kind of *value failure* Bondavalli and Simoncini (1990); McDermid and Pumfrey (1994).

A more subtle failure occurs if the controller of traversal of *newSearch* from *FlightDetails* correctly ends the *enclosed* context but *omits* to begin it again, this is an instance of *LateOrMissingBegin* fault causing a different instance of *vanishing component* failure. Fig. 3.5 illustrates an exemplary sequence of HTTP requests that activates the fault: since all the contextual instances associated with the ended *enclosed* context are destroyed but not re-instantiated, a null pointer exception will occur as soon as any of these contextual instances is invoked; in the example of Fig. 3.5 this occurs immediately, but other user navigation paths might leave the error hidden for a much longer time. Note that this kind of fault will likely escape testing unless some tailored methodology is applied to select input sequences in light of the fragility induced by dependency injection and automated lifecycle management.

Finally, note that the vanishing component failure can be caused by a large variety of further coding faults, beyond the specific defects in the programmatic demarcation of *enclosed* contexts. E.g., a session-scoped dependent component may receive an injection of a required instance whose type is associated with *request* scope, this is the case of a *shorterScope* fault: if the dependent component uses the injected dependency across multiple requests, the reference is not valid and various types of failures may occur. This may happen also if the type of the required component has an *enclosed* scope, which makes detection much more subtle and dependent on the specific navigation path of the user.

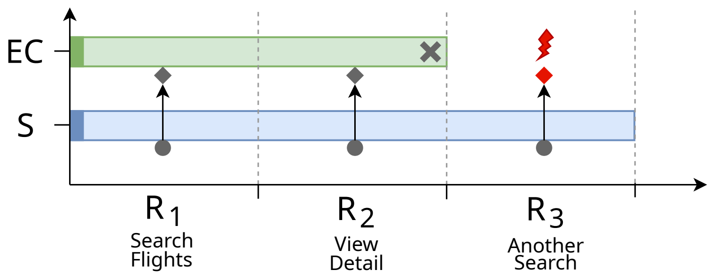


Figure 3.5: Coupling scenario which produces a *vanishing component* failure, both for a visitor and a registered user. In R_1 the end-user starts searching for flights (the *SearchFlightsController* ●, living within the *session* S, depends on the *FlightManagerComponent* ◆, living in the *enclosed* context EC); in R_2 the end-user views the detail of a flight; finally in R_3 the end-user performs a new search (the EC is programmatically closed, then the *SearchFlightsController* tries to invoke the *FlightManagerComponent* which does not exist anymore).

Consider now the case of ToDoApp, as stated in Sect. 2.6.2, the application allows the user to decorate each task with notes. In the specific implementation of this feature, the note management is entrusted to a specific component called *noteManager* that maintains the text of the note as its own state.

The user interface enlists all the tasks on the same page and, inside the space allocated for each task, allows to open and close a *widget* containing a textbox where the note can be visualized and modified.

In this configuration, the obvious solution might be to associate *noteManager* to the visibility context of this latter widget, however doing so the instance of this component will be destroyed as the textbox is closed losing the state of *noteManager*. The correct solution in this case is to associate the component to the visibility context related to the overall task area.

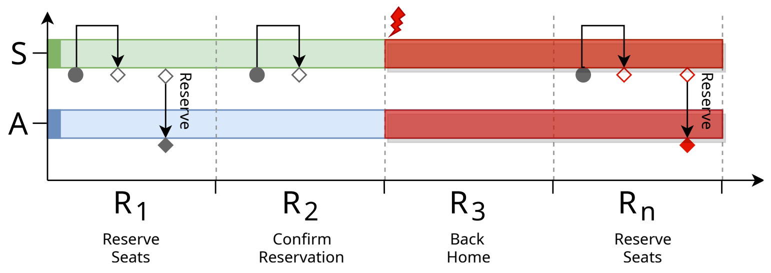


Figure 3.6: Coupling scenario, for a registered user, producing a *zombie component* failure. *TemporaryReservationComponent* ◇ lives within the *session* context S, inheriting from *RegisteredBookingController* ●, and after R_2 it is always in an active context, thus falsifying reservation values of *TemporaryReservationRepository* ◆ within *application* context A.

Zombie components. The failure can be activated during the "Book Flight" use case, identified as UC:7.1, where the procedure is implemented and adapted differently distinguishing between visitor users and registered users. *VisitorBookingController* and *RegisteredBookingController* delegate the seats reservation process to their injected instances of *TemporaryReservationComponent*, living in a *conforming* context, which is designed so as to allocate temporary reservations on demand, releasing them just before it is destroyed (*TemporaryReservationRepository* takes the total count of reserved seats within the whole *application* context).

Note that the two page controllers live in different contexts: the *VisitorBookingController* lives within the *enclosed* context, while the *RegisteredBookingController* lives within the *session* context. The reuse of *TemporaryReservationComponent*, whose lifecycle is inherited by its injector controller, hides a *missingStateClearance* fault for the *RegisteredBookingController* side since the temporary seat release will not be activated until the end of the user session and not immediately after the end of the "Book Flight" use case as expected, this erroneous behavior may produce a *zombie component* failure instance; as represented in the sequence of HTTP requests within the conceptual abstraction of Fig. 3.6 for a registered user.

The same failure can be experienced in ToDoApp, in particular, implementing the “clear project” feature. As already pointed out, each project is made of various tasks managed by a particular component named *taskManager* that maintains all the tasks related to a project.

The user interface of ToDoApp displays on a single page all the information of the project and, a particular bounded area is reserved for a widget that enlists all the related tasks.

If the *taskManager* is programmatically associated with the context related to the task widget, the zombie component failure may occur during the clear project procedure: when the client clicks the button to erase all the tasks the related widget is destroyed, however, the *taskManager* continues to live and maintain the information about the tasks since its life cycle is tied with the project widget. As proof of this, when the task widget is opened again by the user, all the previous tasks will be displayed.

Unexpected shared components. The failure can be activated during the execution of the “Search Flights” use case, identified as UC:U6, 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.

This component is injected at authentication time by the *LoggedUserComponent* (also living in the *session* context), which initializes 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 on the years of affiliation, 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 UC:U6, 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 live within the same long-running *session* context, thus sharing installed dependencies (i.e., *LoggedUserComponent* and *SearchFlightsController* share the *BillingComponent*). So, the application enters an error state which however is not manifested: 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.

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 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. 3.7.

Since the type of fault is strictly related to the provided solution and since this configuration can be corrected in multiple ways, the described fault cannot be classified univocally. The failure could be avoided by assigning the conforming scope to the *BillingComponent* and then preventing the instance sharing among *LoggedUserComponent* and *SearchFlightsController* (this solution classify the fault as a *wrongConformance* fault), but could also be avoided through a specific cleaning of the *BillingComponent* state after the visit of *FlightDetails* page or through an automatic reset in the next instantiation of a dependent component (both faults classified as *missingStateClearance* faults).

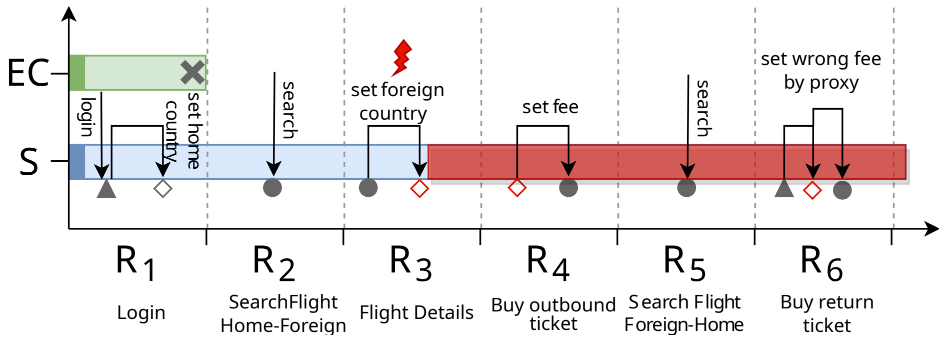


Figure 3.7: Coupling scenario, for a registered user, which produces an *unexpected shared component* failure. *LoggedUserComponent* ▲, *BillingComponent* ◇, and *SearchFlightsController* ● live within the *session* context S and the data of *BillingComponent* are initialised after the authentication process in R_1. After R_4 the system enters a latent error state, considering the unexpected sharing of *BillingComponent* contextual instance.

The Unexpected shared component failure might be introduced also in the ToDoApp: in particular let us focus on the "add priority" use case where the user adds a priority, in a form of an integer from 1 to 10, to a specific task. As already stated, on the project page, tasks are enlisted in a specific area where each one is presented as an embedded widget.

The priority value of the task is managed by a component called *priorityManager* that beyond some additional features, it also maintains the priority of the related task. With this scenario, if the *priorityManager* is injected at the same level of the task list widget, the component will be shared among all the tasks of the project resulting in an unexpected shared component failure: each time the user modifies the priority of a specific task, the application will assign the same priority to all the tasks.

Unexpected injected components. The failure can again be activated during the execution of the "Book Flight" use case, identified as UC:U7.1, and directly affects the case of end-users interfacing with *RegisteredBookingController* which is responsible for 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 for determining the final price of the booking, applying a country fee on the ticket and asking 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 for instantiating at runtime the right discount strategies, also basing the decision on some information maintained within the *LoggedUserComponent* (i.e., on the purchasing history of the current registered user). In this scenario, the failure 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* behavior 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 produced 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 UC:U7.1 use case, *LoggedUserComponent* data may become obsolete, affecting in turn also the programmatic lookup mechanism and classifying the fault as *erroneousDynamicInjection*. The sequence of HTTP requests leading to the fault occurrence is represented in the conceptual abstraction of Fig. 3.8.

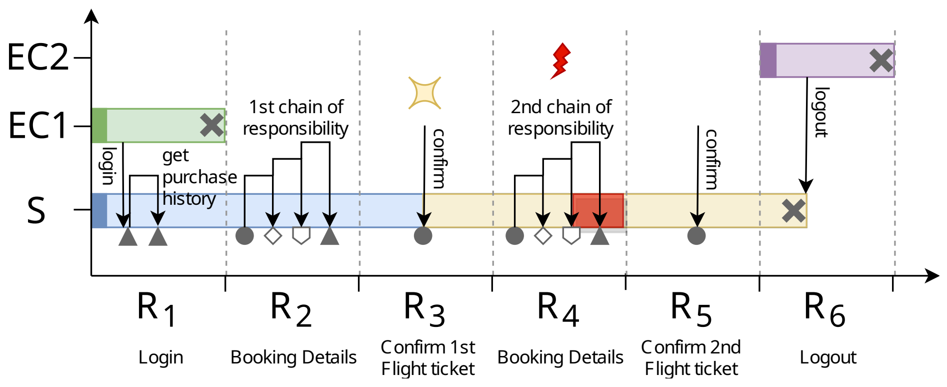


Figure 3.8: Coupling scenario, for a registered user, which produces an *unexpected injected component* failure. Within the *session* context *S* there are three managed components (i.e., *RegisteredBookingController* ●, *BillingComponent* ◇, and *LoggedUserComponent* ▲) and by design they establish 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 of dynamic injection, implemented in *DiscounterComponent*, is disrupted after R_2 for the whole end-user session for a data inconsistency (i.e., ✧) induced on *LoggedUserComponent*.

3.4 Fighting Faults through Model-Based Testing

Searching defects in the specification of managed components lifecycle and in their composition requires that testing be focused on input sequences that can activate faults and let them propagate up to produce a failure in the behavior delivered by the user interface or in some observable level of components state. This subtends the ability to identify input sequences that cover component dependencies by reproducing actual conditions of concurrency in the logic of intervals during which managed components maintain their state.

To this end, it is proposed model-based testing approach Utting et al. (2012) that jointly involves the constraints of the page navigation diagram of the user interface, the lifecycle specification of back-end components and their data-flow dependencies, and the actual concurrency produced by the effects of container orchestration.

The approach relies on an abstraction called here Managed Components Data Flow Graph (*mcDFG*, Sect. 3.4.1), which supports the identification of test suites implementing various data flow coverage criteria (Sect. 3.4.2), and which can be derived from basic artifacts of the development process through a disciplined approach, with automation of the most expensive and error-prone transformation steps (Sect. 3.4.3).

3.4.1 The Managed Components Data Flow Graph 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 run-time data flow produced by *active* contextual instances. In principle, these paths might be abstracted into an *Object-Oriented Data Flow Graph* Souter et al. (1999). However, this would require explicit unfolding and representation of the complex actions performed by the DI container in the management of contextual instances (e.g., components proxies, aspect-oriented programming techniques), with an explosion of graph elements leading to an infeasible dimension of test suites.

To this end, the *Managed Components Data Flow Graph* (mcDFG) abstraction is proposed, inspired by the classical DFG and DFT theory Rapps and Weyuker (1985), which combines elements of structural and functional perspectives by capturing salient characteristics of involved components with their dependency hierarchies and lifecycles together with admissible end-user navigation and interactions along designed use cases.

Formally, the *mcDFG* is a directed graph, labeled on vertices and edges $mcDFG := \langle \mathcal{V}, \mathcal{V}_{in} : \mathcal{E}, \mathcal{E}_{in}, def, use, \mathcal{P}, Nav, CB \rangle$, where:

$\mathcal{V}$ is the set of vertices, with each $v \in \mathcal{V}$ representing a *basic block*, i.e. a sequence of method invocations and DI container instantiations that are always executed as a whole; $\mathcal{V}_{in} \subseteq \mathcal{V}$ is the subset of vertices associated with basic blocks that terminate in any state where the user interface waits for user input;

$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, with $< v_i, v_j > \in \mathcal{E}$ iff there exists an execution where the last operation of v_i can be followed by the first operation of v_j; $\mathcal{E}_{in} \subseteq \mathcal{V}_{in} \times \mathcal{V}$ is the subset $\mathcal{E}$ made of the edges that leave a basic block that terminate with the interface waiting for user input.

relations $def : \mathcal{V} \to 2^{MC}$ and $use : \mathcal{V} \to 2^{MC}$ associate each vertex with the subset of *used* and *defined* managed components, where MC denotes the set of all managed components, and, for any $c \in MC$, $c \in def(v)$ means that an instance of component c is created during the execution of the basic block associated with vertex v, and $c \in use(v)$ means that an already existing instance of c is used by the invocation of any of its methods; note that, as opposed to the classical theory of dataflow testing, the relation of *use* does not distinguish whether the invocation will produce a side effect on the used component; besides, the relation $\mathcal{P} : \mathcal{V}_{in} \to pages$ associates each vertex $v \in \mathcal{V}_{in}$ with the page displayed on completion of its associated basic block;

the relation $Nav : \mathcal{E}_{in} \to \{nav\, page\, controller :: sign()\}$ associates each edge $\epsilon \in \mathcal{E}_{in}$ that exits from a vertex $v \in \mathcal{V}_{in}$ with the page controller method invoked on the input of the user action $sign()$; the relation $CB : \mathcal{E} \rightarrow EnclosingActions$ associates edges with any programmatic action of control of an *enclosed* context performed when the edge is traversed, with $EnclosingActions = \{begin, end, end/begin\}$.

To exemplify the concept, Fig. 3.9b reports the mcDFG for the Admin use case *"View Airports"* (UC:A4.2): vertices, associated with basic blocks, are represented as green circles and they are labeled with def and *use* operations performed in the corresponding basic block, on a violet and green background, respectively (e.g. see, vertex 1); vertices in $\mathcal{V}_{in}$ (e.g. vertex 5) are also associated with a pale blue label with the name of the page where the interface waits for user input; output edges from $\mathcal{V}_{in}$ vertices are labeled with the name of controller methods invoked on user actions available in the input page (e.g. from vertex 5, AirportController::viewAirport() and AirportController::redirectToHome()) actions for programmatic control of enclosed contexts are labeled on edges where they occur (e.g. on edges $\langle 1,2 \rangle$ and $\langle 3,0 \rangle$).

Note that the mcDFG is a kind of grey-box abstraction that seams the structure of the page navigation diagram of Fig. 3.9a (the pale blue parts) together with lower-level information related to the application code (green parts) and the DI container behavior (violet parts).

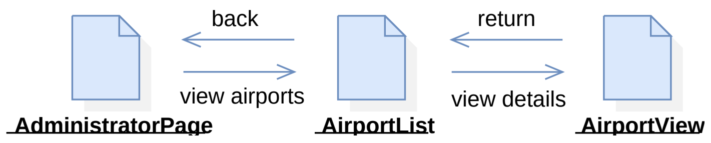
3.4.2 Test Case generation based on the mcDFG

The mcDFG abstraction captures couplings among managed component instances under the orchestration of the DI container according to actions taken by the user in the navigation of interface pages. Coverage of these coupling comprises a focused and effective means for the identification of faults in annotation-based and programmatic DI specification of back-end components. In so doing, a feasible *mcDFG* path subtends a sequence of user interactions on the UI that triggers a specific chain of interactions among managed components. A single path is embedded in a test case and the set of paths satisfies a certain coverage criterion in a dedicated test suite.

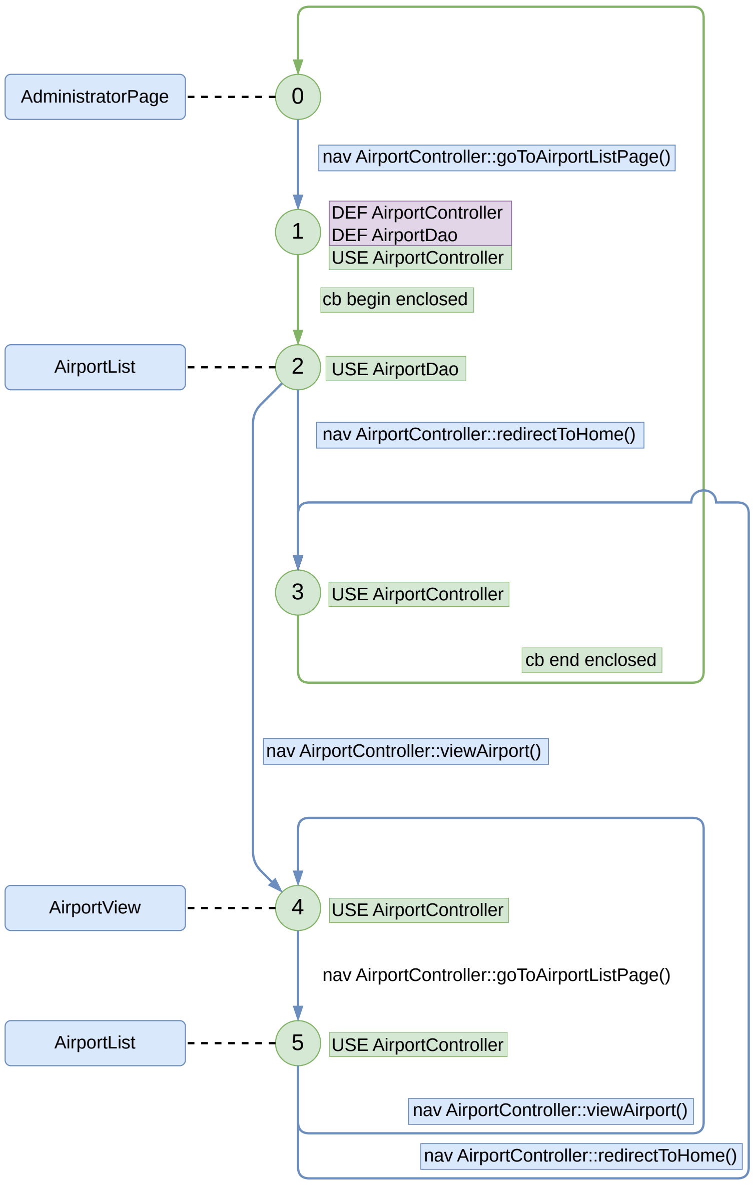
Note that, a test case identified by the *mcDFG* abstraction implies a navigation constraint to verify the actual implementation and so, the path identification phase also defines a base oracle, open to be extended by the tester through specific inspections on the state of both the user interface and the business logic.

Without loss of generality, it is considered a suite of criteria inspired by the classical theory of Data Flow Testing Rapps and Weyuker (1985), while various other coverage criteria could be implemented as well (e.g., page and hyperlink testing as described in Ricca and Tonella (2001)).

- **All Nodes** coverage verifies that every reachable basic block is tested at least once, which includes that each *def* (i.e., a component instantiation) and each



(a) A fragment of the Page Navigation Diagram.


AdministratorPage
0
nav AirportController::goToAirportListPage()
1
DEF AirportController
DEF AirportDao
USE AirportController
cb begin enclosed
AirportList
2
USE AirportDao
nav AirportController::redirectToHome()
3
USE AirportController
cb end enclosed
nav AirportController::viewAirport()
AirportView
4
USE AirportController
nav AirportController::goToAirportListPage()
AirportList
5
USE AirportController
nav AirportController::viewAirport()
nav AirportController::redirectToHome()

(b) Managed Component Data Flow Graph.

Figure 3.9: A snippet of PND and the corresponding mcDFG for the administrator use case “View Airports” (UC:A4.2).

use (i.e., a component method invocation) of any managed component is exercised;

- **All Edges** verifies that every edge is traversed at least once, which implies that each *nav use* from each page (i.e., each end-user interaction) is tested;
- **All Defs** verifies that every *def* is tested at least one time, thus exercising each managed component instantiation, reaching one of its *uses* (i.e., one of component method invocations), without traversing intermediate *defs* of the same component;
- **All Uses** verifies that for each *def* all the possible subsequent *uses* are covered, i.e. that: for each component c, and each vertex v_d where c is *def*ined, and each vertex v_u where c is *use*d, *at least one path* that goes from v_d to v_u without visiting any intermediate *def* is exercised;
- **All DU-Paths** verifies that all the possible acyclic paths between each *def* and all its subsequent *use*s are covered, i.e. that: for each component c, and each vertex v_d where c is *def*ined, and each vertex v_u where c is *use*d, *all the acyclic paths* that go from v_d to v_u without visiting any intermediate *def* are exercised.

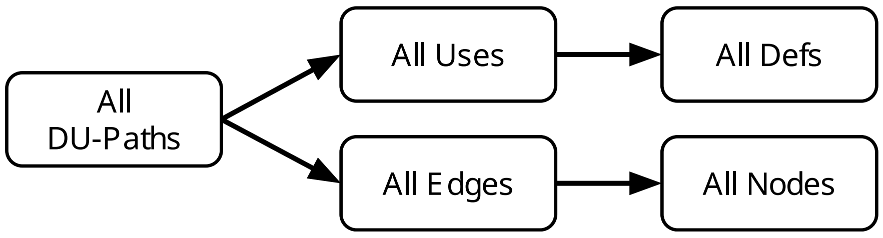


Figure 3.10: Inclusion relationships among coverage criteria for the *mcDFG* abstraction.

Inclusion relationships among different criteria are summarized Fig. 3.10. Note that they differ from those of the classical theory of data flow testing in Rapps and Weyuker (1985) in that *All Uses* coverage does not include *All Nodes* (and not either *All Edges*): in fact, in the mcDFG, branching edges from a basic block represent user choices in navigation control, not alternative complementary exits of a common guard expression as leveraged in the proof of coverage inclusion referred to the Data Flow Graph in Rapps and Weyuker (1985).

Theoretical complexity, expressed in terms of the limit number of tests sufficient to implement each criterion, is reported in Tab. 3.1, where N is the number vertices in the *mcDFG* abstraction, C the number of distinct managed components, and F the maximum number of user choices in the navigation out of any page within a use case

Criterion	Complexity
All Edges	$\mathcal{O}(N \cdot F)$
All Nodes	$\mathcal{O}(N)$
All DU-Paths	$\mathcal{O}(2^N)$
All Uses	$\mathcal{O}(N^2)$
All Defs	$\mathcal{O}(N \cdot C)$

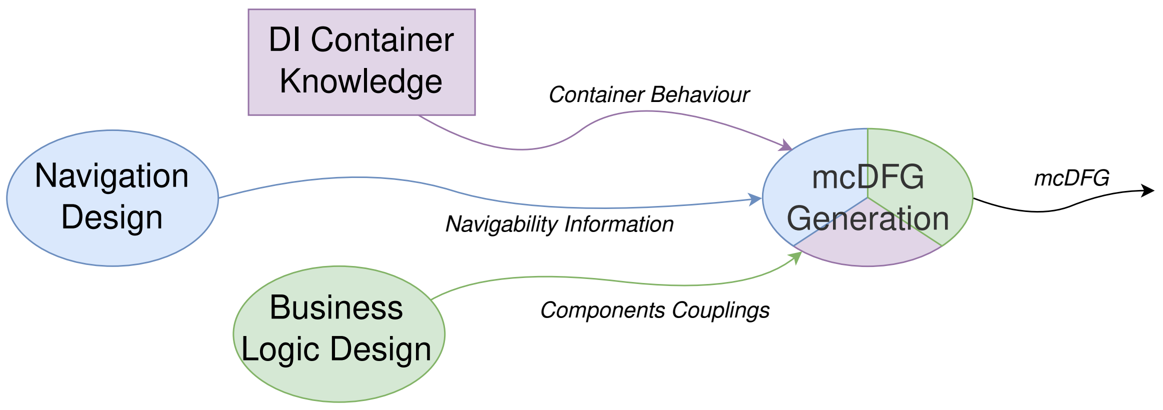
Table 3.1: Complexities of *mcDFG* coverage criteria.

3.4.3 mcDFG generation

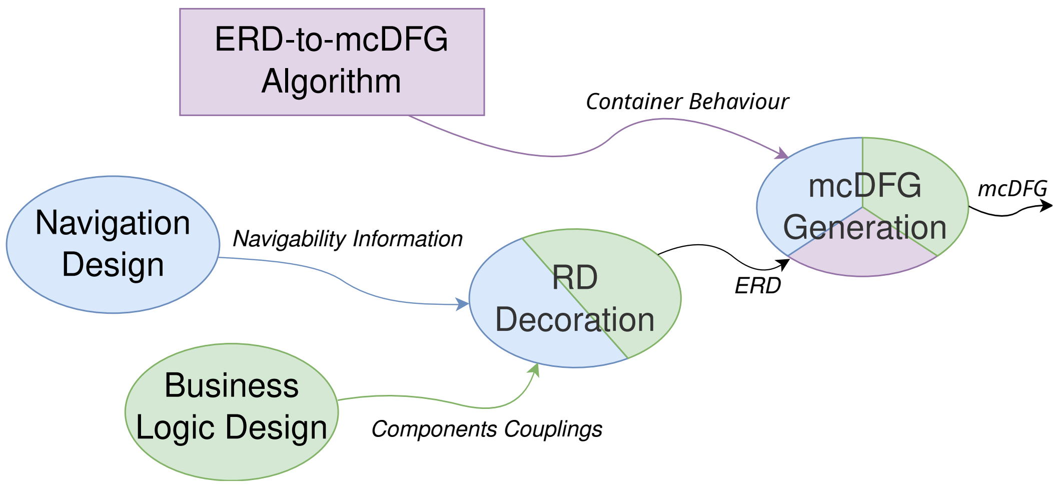
The mcDFG provides a powerful abstraction, well-tailored to unravel the actual dependencies that result from the intertwined effects of user navigation of interface pages, DI specification and method invocations in back-end components, and orchestration mechanisms implemented by the container. However, this effectiveness comes with a corresponding price in the mcDFG construction, which involves a significant and error-prone effort for the inherent complexity of integration of different perspectives and for the possible misconception of the DI container behavior (see Fig. 3.11a).
To overcome the hurdle, it is proposed a model transformation approach (outlined in Fig. 3.11b) based on an intermediate artifact, termed *Enriched Robustness Diagram* (ERD), which is then used for automated generation of the mcDFG. In this process, the ERD remains completely unaware of the DI container behavior and can thus be obtained with simplicity by the decoration of the basic Robustness Diagram, so that most of the effort and error-proneness, and also the dependence on the specific DI technology, are absorbed by the automated step of model transformation.

The Enriched Robustness Diagram (ERD) decorates the basic Robustness Diagram with additional information about the structure of the business logic, which can be conveniently expressed through *ad hoc* stereotypes or action edges, as illustrated in Fig 3.12 for the RD of Fig 2.11:

- each *controller* element is decorated with its context and its invoked primary method (e.g., in Fig 3.12, the *AirportController* is bound to an *enclosed* context and, on the navigation from *AdministratorPage* to *AirportList* page, its *goToAirportListPage*() method is invoked).
- method sub calls are repeatedly expanded and represented by usage relationships (dashed edges) among *controller* entities, with labels marking the invocation order; as a part of this, also programmatic actions on *enclosed* contexts boundaries are determined and annotated with labels of type *EnclosedContext::begin*() and *EnclosedContext::end*() (e.g., the *goToAirportListPage*() method includes a sequence of sub calls made of the begin of an *enclosed* context and



(a) Manual mcDFG generation process.



(b) mcDFG generation process through ERD-to-mcDFG algorithm.

Figure 3.11: Comparison between the manual mcDFG generation process (a), and the automated generation through the automated transformation from the intermediate ERD model (b).

the invocation of an *AirportDao* method, executed in this order as shown by labels #1 and #2.).

Note that the production of the ERD is conceptually simple, and it can be easily accomplished manually by static inspection of salient information in the code of view pages, page controllers, and referenced managed components, even by a developer without strong knowledge of DI containers. In principle, this could also be generated automatically through automated source code *static analysis*.

The mcDFG can then be derived from the ERD by adding the details related to the DI container mechanisms. This is performed automatically by merging the information

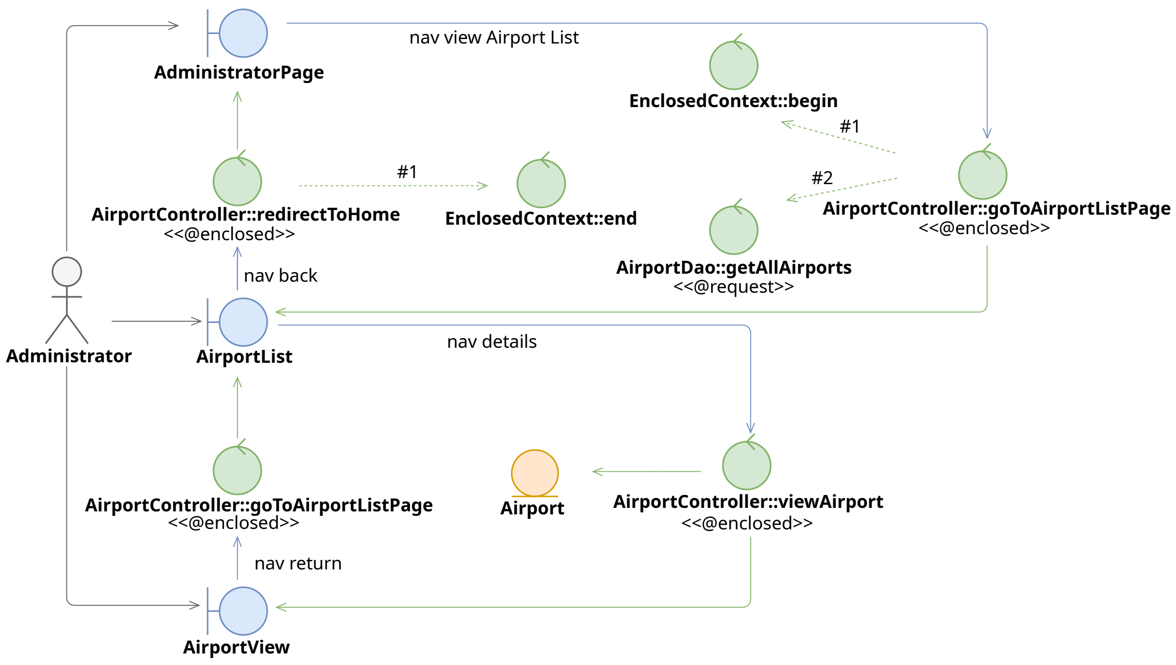


Figure 3.12: Enriched UML Robustness Diagram of “View Airports” use case (UC:A4.2) for an admin user, shown in Fig 2.11.

about application pages navigability and component dependencies provided by the ERD in input with the characteristics of the behaviour of the DI container (See again Fig. 3.11b). For this work, this automated transformation was implemented for the case of the CDI container of the JEE Architecture. The implementation works iteratively starting from an initial boundary object of the ERD and, exploring all possible actions that a user could undertake (i.e., outgoing edges), it transposes them into edges and basic blocks of the mcDFG. As notable features, the algorithm optimises the number of final basic blocks and implements heuristics that keeps the number of cycles as low as possible, with a positive impact on the number of paths that shall then be covered by different test cases.

Finally, note that the usage of the ERD as an intermediate abstraction provides a manifold benefit: *i*) it speeds up the process since the ERD can be generated through a simple decoration step of the robustness diagram or through source code static analysis, *ii*) it prevents the developer to inject errors related to the DI container in the mcDFG since this knowledge is encapsulated in the algorithm that represents a “single point of container knowledge dependency”, *iii*) finally as a consequence of this latter point, it allows to obtain mcDFG referring to different DI containers implementation without effort by only changing the algorithm implementation itself while maintaining the same ERDs.

3.5 Experimentation

This section reports experimental results showing how the mcDFG provides an effective abstraction for the selection of test cases that are able to: activate faults occurring in the usage of dependency injection and automated management of components lifecycle; and propagate them up to failures in the functional behaviour of the user interface or in some observable inconsistency of the state of business logic components.

Specifically, the experimentation process and the obtained results are drawn in Sect. 3.5.1 and Sect. 3.5.2 respectively and finally, validity threats are discussed in Sect. 3.5.3.

3.5.1 Experimentation protocol

A suite of faulty mutations of the *Flight Manager* application of Section 2.6.1 was tested using a variety of test suites covering the *mcDFG* abstraction according to different criteria.

For each use case, the *mcDFG* was derived automatically from the corresponding *Enriched Robustness Diagram* using the ERD-to-mcDFG model transformation described in Section 3.4.3. On each *mcDFG* and for each of the coverage criteria discussed in Section 3.4.2 (i.e. *All Nodes*, *All Edges*, *All Defs*, *All Uses* and *All DU Paths*), a test suite was identified as a set of *mcDFG* paths, each comprising a test case. Each test case was also associated with assertions about post-conditions for the expected appearance of the frontend (*e.g.*, after a given user navigation sequence only certain items should be displayed in the list) and for the expected inner state of managed components (*e.g.*, at the end of the navigation sequence, the field *field1* of the instance *instanceA* should have a given value).

The correct baseline of the *Flight Manager* was then mutated to produce 32 faulty versions by manually injecting faults so as to cover the classification of fault types of Sect.3.3.1 and to produce one of the failure modes of Sect 3.3.2 under some navigation sequence occurring in realistic common executions of addressed use cases.

Four instances of the 32 faulty versions are described in Section 3.3.3, and complete documentation is provided in the GitHub repository, which thus includes $1 + 32$ versions of *Flight Manager*: the branch *main* of the repository contains the correct baseline version, while each of the other branches contains a different faulty version obtained by manual injection of a type of fault causing one of the addressed failure modes, each accompanied by respective diagrams and artifacts. Note that this comprises a significant base of code and accompanying artifacts open to further experimentation beyond the objectives of this work. In particular, the correct baseline version is documented with both standard artifacts i.e., class diagram, deployment diagram, use case diagrams and page navigation diagram and specific artifacts i.e., Enriched Robustness Diagrams, corresponding managed component Data Flow Graphs and the test suites of each coverage criteria. Besides, each faulty version is further decorated with an ERD that highlights the fault instance and the corresponding test suites that are capable or not to detect the fault. [1]

The test suite of each coverage criterion was finally run against each of the *Flight Manager* faulty versions, to check whether each fault is revealed by the failure of at least one assertion in at least one test case. In so doing, the fault detection capability of the underlying coverage criteria is assessed, and, indirectly, the effectiveness of the mcDFG abstraction is validated.

To further evaluate the proposed methodology, it is compared to the fault detection capability against coverage criteria that can be implemented in end-to-end functional testing directly based on the page navigation diagram (*PND*, see Fig. 2.8). Specifically, it is considered *All Pages* coverage, which requires that each reachable page is visited at least once, and *All Navigations* coverage, which verifies that each navigation (i.e., each edge of the page navigation diagram) is traversed at least once.

3.5.2 Results and Discussion

Abstraction	Coverage Criterion	Test Suite Dimension	Interactions per Test Case	Fault Detection Capability (%)
mcDFG	*All Nodes*	1.18	6.09	100
	All Edges	1.27	9.25	100
	All Defs	1.18	3.09	84.37
	All Uses	2.27	5.04	100
	All DU Paths	3.09	7.76	100
PND	*All Pages*	2	18	28.12
	All Navigations	3	26.33	50

Table 3.2: Complexity and fault detection of coverage criteria on the 32 faulty versions of *Flight Manager*.

Table 3.2 summarizes experimentation results showing complexity (Average number of Test Cases and interface interactions per Test Case) and detection capability of different coverage criteria based on the *mcDFG* and the *PND*.

All coverage criteria based on the *mcDFG* show a high fault detection capability, full in most cases, and definitely over-perform test suites based on the *PND* abstraction. This improvement can be explained as due to the ability of the *mcDFG* to extend the purely functional perspective of the *PND* with architectural information, which

[1] A guide to the structure of the repository is also included in the additional material (Chapter A)

supports both test case selection and oracle interpretation: on the one hand, test cases identify navigational paths that stress the application not only under the end user functional perspective of pages navigation but also under the business logic and DI container structural perspective; on the other hand, test cases and interpretation of their effects are built so as to be aware both of the user interface and of the business logic components states, enabling detection of a fault even when its propagation does not manifest a failure at the user interface and remains hidden with consequences that are hard to observe and predict Grottke and Trivedi (2007).

It is worth noting that *All Defs* performs worse than all the other criteria, which can be explained as a consequence of the fact that *All Defs* coverage can be implemented by extremely compact paths, where some component methods may not be exercised at all, as illustrated in Fig. 3.13.

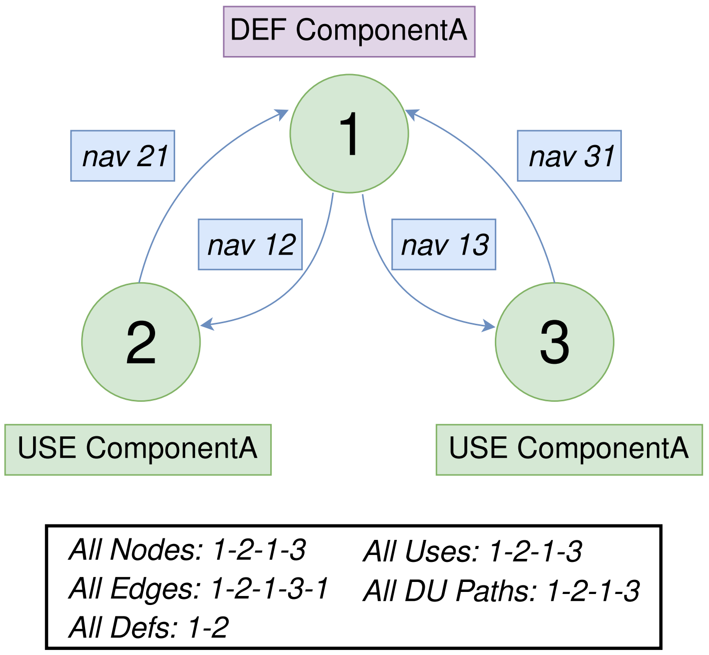


Figure 3.13: Different coverages on a specific *mcDFG* example.

The size of *mcDFG* test suites, expressed as the average number of test cases requested for implementation of each criterion, remains low and definitely affordable even for expensive criteria, and notably for *All DU Paths*; this depends on the fact that the *mcDFG* is a high-level abstraction resulting in a sparse graph with a limited number of vertices and edges, related by construction to the number of pages and actions involved in each use case, by far lower than what may occur in a conventional DFG expressed in terms of code-level basic blocks.

In the comparison of dimensions of *mcDFG* and *PND* test suites, it is worth noting that the value related to the *mcDFG* represents the average number of test cases needed to satisfy the coverage criterion in a use case, while the *PND*-related is the exact number of test cases needed to test the entire application. In fact, each methodology was used in its natural way: *mcDFG*-based testing is use-case-wise, as it identifies a different suite for each use case, while *PND*-based testing targets the interface pages of the overall application, which can be covered with a limited number of "long" test cases. However, even if the dimension required to test the *entire* application with the proposed methodology is still low (the larger test suite is the one related to the *All DU Paths* criterion and consists in 20 test cases), it is possible to include multiple use cases in the same *mcDFG* and then exploit the connectivity between pages to further decrease the test suite dimension. This kind of trick, however, has a drawback: while the number of test cases decreases, due to the redundant navigation actions used, the length of each individual test case (i.e., the number of user interactions required to carry out the selected navigational path) increase, suggesting that the test suite execution time will not change too much with the use case wise or the application-wide approach (see the number of interactions per test case in the Table). As a showcase, an *mcDFG* comprising both the "*search flights*" and the "*book flight*" use cases (UC:U6 + UC:U7.1) was also generated obtaining test suites with the same fault detection capability obtained with the two separate diagrams, with an overall smaller size and as expected, with longer sequences characterizing each test case (see the details in the repository).

3.5.3 Threats to Validity

The proposed methodology was experimented on a specific Web application (Flight Manager), based on specific architectural patterns (1), developed on a specific language (Java) and technological stack (JEE) (2), by software professionals connected with the same Lab where the proposed methodology is elaborated (3). In principle, this may jeopardize *external validity*, as a different architecture might be adopted (1), other languages and stacks be used (2), and developers might be prone to different types of faults hidden by a bias due to a specific level of skill and experience (3). To limit these threats, various countermeasures were assumed.

(1) *Flight Manager* implements a widespread combination of reference architectural patterns, largely documented in the professional literature Richardson (2006); Martin (2017); Fowler (2012), and (2) developed using a language and technology stack (Java and JEE) with primary impact and spread in the practice of complex web applications.

Besides (3), types of Faults reproduced in the *Flight Manager* were not limited to the accumulated experience and current level of skill of the Lab, but they also considered the inherent structural complexities of annotations and a collection of oddities about components lifecycle management reported in technical social forums (*e.g., StackOverflow* and *GitHub*) by developers with different levels of experience and different expertise in language and frameworks. [2].

Moreover (2,3), the types of faults addressed in this work refer to general mechanisms of dependency injection management for which the proposed methodology provides a general abstraction independent of the specific architecture and stack of technologies of the application under test.

It is also worth stressing that (2), while experimentation and presentation refer to a specific programming language and technology, artifacts and navigation sequences that fulfill each of the coverage criteria would not substantially change in the implementation of the same design on a different technological stack.

[2]A collection of discussion threads addressing this class of difficulties is included in the additional materials

Chapter 4

Using Life Cycle Management as Software Rejuvenation

In the development of software architectures application logic, the scope of the managed components is specified by the developer through class-level directives, which define the rules of a choreography that determines when the state of components will be refreshed. More specifically, this defines a kind of *micro-rejuvenation* Sundaram et al. (2008); Avritzer et al. (2020), that does not require reboot of the physical servers Koutras and Platis (2006), of the *Virtual Machine* Machida et al. (2010), of the *Application Server* Alonso et al. (2013), or of the client-side mobile device Cotroneo et al. (2016); Xiang et al. (2019), and it is instead obtained by restarting instances according to the specified scope of their types.

In general, the same functional behavior of the user interface can be obtained through different scopes, giving the developer the choice as to whether managed objects shall live for a shorter or longer lifecycle. This gives raise to a design space where the developer defines a kind of rejuvenation policy: shorter scopes will result in a more frequent refresh of the state of managed objects, limiting the number of paths through which errors can propagate across objects with overlapping activity cycles; conversely, longer scopes maintain alive instances in-memory for a longer time, limiting memory operations and database load, at the price of a higher exposition to error propagation and aging processes Cotroneo et al. (2011, 2014).

In this Chapter, it is characterized how the session state maintained in the application logic of software architectures may host and propagate errors, and how this threat is contrasted by the mechanism of micro-rejuvenation produced by automated management of components' life cycle.

To this end, the impact of micro-rejuvenation on the propagation of errors is made observable through an experimental study, supported by a suite of novel tools, open to reuse and extension beyond the specific objectives of this dissertation: a small-sized exemplary web application that reproduces the common architectural patterns of a stateful web application based on the stack of Java/Jakarta Enterprise Edition (JEE); a software fault injector Madeira et al. (2000); Natella et al. (2012) that supports accelerated testing Meeker and Escobar (1993); Escobar and Meeker (2006); Limon et al. (2017) by emulating the arrival process of faults activations in the components of the business logic of any third-party web application; a monitor that logs the state of the business logic of any third-party application so as to identify the structure of epochs during which injected faults can propagate across running components with overlapping life cycles. Experimental results qualitatively confirm that different design choices have an impact on the resilience to faults due to the impact of different frequencies of micro-rejuvenation of components. Automated reconstruction of duration and complexity of epochs during which components overlap their life cycle permits to explain observations by highlighting paths of error propagation.

In the rest of the Chapter, first, it is characterized the mechanism of micro-rejuvenation produced as a side effect of lifecycle management (Sect. 4.1), and then it is described the setup and tools implemented for the experimentation (Sect. 4.2), finally obtained results are drawn in Sect. 4.3).

4.1 Micro-Rejuvenation by lifecycle management

When specifying the lifetime associated with different component types, the developer defines a choreography that will be enforced by the DI container during the runtime. Different choices can produce the same functional behavior observed at the user interface but with a different evolution of the *session state*, which results from the composition of the states of running instances. In particular, this changes the logic of intervals Allen (1983) during which instances live, and thus impacts the structure of paths through which errors can be maintained in memory and propagate across components up to possibly produce a failure.

In this perspective, the choice of scopes implicitly specifies a kind of *software rejuvenation* policy that will be implemented by the *Container* through automated handling of creation, initialization, and destruction of managed components (See Fig. 4.1).

Note that this will not result in a reboot of the physical servers (hot/cold spares) Koutras and Platis (2006), of the *Virtual Machine* Machida et al. (2010), of the *Application Server* Alonso et al. (2013), or of the client-side mobile device Cotroneo et al. (2016); Xiang et al. (2019). Instead, this comprises a reboot at the software component-level that produces a kind of *micro-rejuvenation* Sundaram et al. (2008); Avritzer et al. (2020). In this perspective, wider scopes, maintain alive (in-memory) instances for a longer time and thus expose components to aging processes Cotroneo et al. (2011,

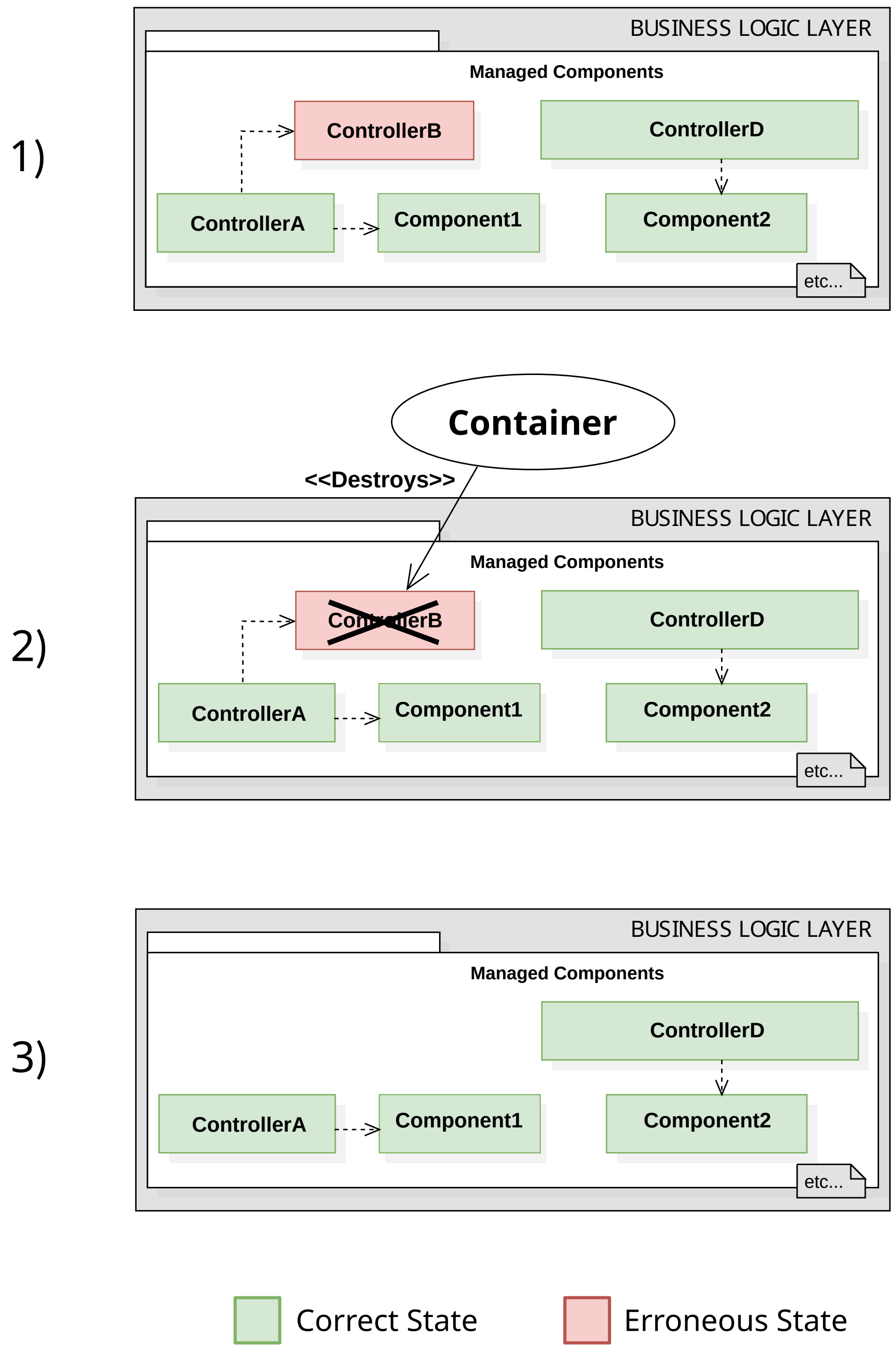


Figure 4.1: Software micro-rejuvenation process executed by the DI container. In *1*) the session state has an error since the component instance `ComponentB` in an erroneous state, in *2*) `ComponentB` ends its life cycle and the DI container destroys the instance, after the destruction, step *3*), the session state is returned correctly.

2014), while shorter scopes produce more frequent refresh of the state of each single component instance, thus limiting the Age of Information and the probability of failures Menasché et al. (2019).

For instance, in the field of stateful web application by associating each managed type with a *request* scope, the developer maximizes the frequency of rejuvenation, which will occur for each component on completion of the actions triggered by each HTTP request generated by the User Interface (UI). This minimizes the probability that an error, occurred in the state of some managed component, is propagated in the computation and then transferred, either to other components or to functional behavior delivered by the UI. However, components living only for the time of a single request result in *stateless* application logic and thus require that data needed along the user *session*, the session state, to be stored with some more resource demanding action, usually implemented through a DBMS access or through web cookies Moore and Freed (2000)).

Conversely, if a component is associated with the *session* scope, any error accumulated along its interactions may be maintained and propagate along the Fault-Error-Failure (FEF) chain Avizienis et al. (2004) established by dependency relationships among components. as a logical consequence, the aging effects may be reduced by *conversation*-scoped components and may be maximized by *application*-scoped components, that maintain and propagate their states until the application shut down.

4.2 Fault Injection and State Monitoring

This section describes an experimental setup developed to inject errors in the session state of a web Application following the stateful architectural style and to observe their propagation during user interaction.

To this end, initially, responsibilities and collaborations of the modules that compose the experimental setup are outlined (Sect. 4.2.1), then they are provided structural details on the design and implementation of the most complex modules (Sect. 4.2.2), finally, observations that modules permit to obtain are characterized in Sect. 4.2.3.

4.2.1 Participants and collaborations

The experimental setup relies on a *framework* made of 4 modules:

- a stateful Web Application, which comprises the Implementation Under Test (IUT), which is equipped with a *suite* of usage *scenarios*, each identifying a navigation path across the pages of the User Interface, i.e. a sequence $\{\langle pag_n : act_n \rangle\}_{n=0}^{N}$ where pag_n is the n-th page visited and act_n is the user action that leads the application from pag_n to pag_{n+1};

- a Web Driver that is able to mock the behavior of a real user by replacing his/her manual interaction with the IUT by applying an action among those that are accepted in the current page of the User Interface (e.g., click a button, fill a form etc. . .);
- a Fault Injector that is able to identify the currently living components in the Business Logic of the IUT and to determine and apply a perturbation on their state;
- an Orchestrator, which repeatedly selects a scenario, determines the sojourn times in each traversed page and the time when a fault must be injected and then performs the experiment by invoking the Web Driver and the Fault Injector to reproduce the scenario flow and inject the fault.

Figure 4.2 represents how the Fault Injector and the Web Driver interact with the stateful web application.

Note that, *when* and *how* the application state is perturbed during the scenario execution implicitly defines the type of reproduced faults. The framework allows agile customization of both aspects (as detailed in Sect. 4.2.2);

Without loss of generality, experimentation reported here was focused on failures arriving *over time*, independently from the specific actions taken by the user and by the application. As a notable case, this can result from *soft errors* affecting transient and persistent memories (e.g., RAM, cache, hard disks) May and Woods (1978, 1979); Baumann (2005). This class of errors was widely addressed in the literature of dependability, in machine learning techniques based on (deep) neural networks dos Santos et al. (2017); Azizimazreah et al. (2018), in availability and reliability analyses of system-level effects over data storage systems Kishani et al. (2019), and more recently in the specific evaluation of sensitivity different RESTful frameworks Cerveira et al. (2020). Various other conditions may result in the arrival over time of errors due to external causes, including transient overloading of the application server, limited resources or environmental noise in IoT edge computing systems Andrade and Machida (2019).

For this reason, the time elapsing between the navigation interactions that characterize the real usage of the IUT, assumes an important role in the proposed experimentation, it has been decided then, to characterize each interaction with a random sojourn time controlled by a tailored distribution, in this way it is possible to mark each navigation step with a different amount of wait time that concretely takes the form of a sleep time between the driver interactions; besides it has been represented also the external failure activation time with proper distribution. In so doing, different execution of the same navigation path will present different situations influenced

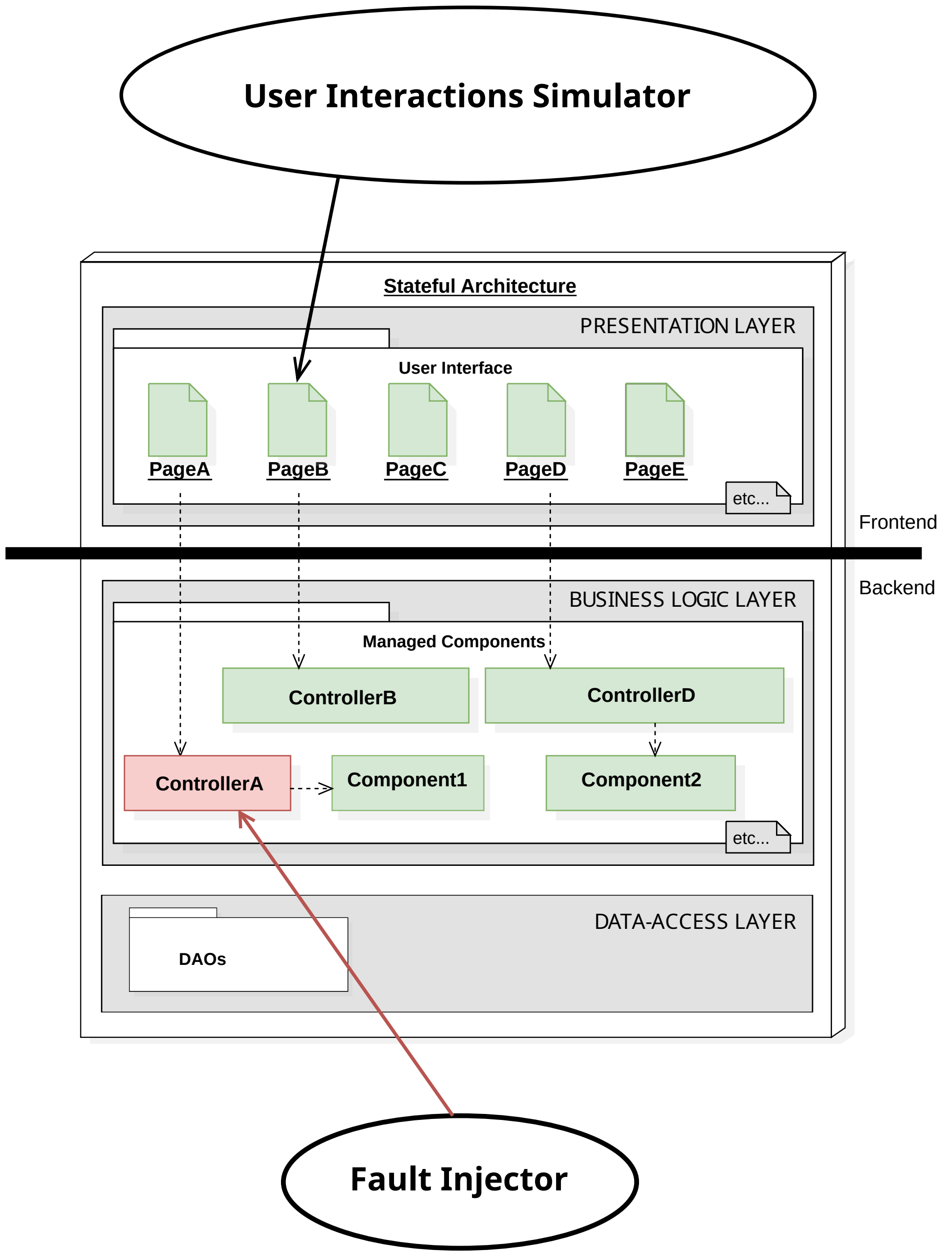


Figure 4.2: The concurrency behavior of the Fault Injection and the Web Driver on the IUT.

by both the time waited between each interaction by the mocked user and both the failure activation time.

In the experimentation, page sojourn times between subsequent requests were exponentially distributed, while faults' arrival times were distributed uniformly within the total duration of the timed scenario. Note that the latter assumption of uniform distribution comprises a fairly precise approximation for the case where: errors arrive according to a Poisson process with a constant rate much lower than that of the sojourn time in each visited state; and, the evaluation is limited to observe only the tests hit by at least one arrival. Under these conditions, the probability of multiple errors during the same test case is negligible, and the arrival time is distributed according to a truncated exponential that can be closely fit by a uniform distribution.

The above presented configuration allows to faithfully reproduce a situation where an error is activated during the usage of the IUT, however, to obtain solid observation insights, each usage *scenarios* of the *suite* should be executed multiple times, for this reason, the presence of various sojourn times in each execution prevents results from being achieved within a reasonable amount of time. To minimize the required time to execute a navigation path, active waits after each driver step are avoided and in their place are used wait times extracted from the interaction steps distributions combined with the error activation time, to only identify before which step the fault injector tool should be triggered.

According to this, the Orchestrator executes a scenario: *i*) by sampling sojourn times of subsequent steps; *ii*) by extracting the *i*-th step to perform the injection following the probability distribution proportional to the sojourn times; *iii*) by executing the web driver without actively waiting the sojourn times until the *i*-th step is reached; *iv*) by injecting the fault through the fault injector; and finally *v*) by executing the *i*-th and the remaining navigation steps through the web driver without actual wait times. In so doing, an event-driven simulation is enabled where the time is not linearly accelerated by "waiting" and "sleeping" mechanisms so as to speed up the flow of time, but it is continuously carried forward to the instant corresponding to the nearest future event (i.e., the lowest sample).

Note that this "acceleration" technique can be applied without loss of generality since the application state changes only in response to a request which is triggered by a user interaction, thus, an external failure manifested between two interactions, will not show any side effect until the next interaction.

4.2.2 Structure and implementation

All the tools were implemented in the JEE ecosystem, leveraging Java Reflection and the *Service Provider Interface* (SPI) of the CDI specification, in integration with the Arquillian Framework Ament (2013), in particular, *Arquillian Drone*, an extension of *Selenium Web Driver*), ShrinkWrap and Arquillian Warp. In the sequel subsection, major software design choices that permitted the implementation of the Fault Injector and the Orchestrator modules are given.

Fault Injector

The Fault Injector is implemented as a *framework* (see Fig. 4.3) using the Factory method (Strategy) pattern permit agile adaptation of aggregated classes `InstanceFinder`,

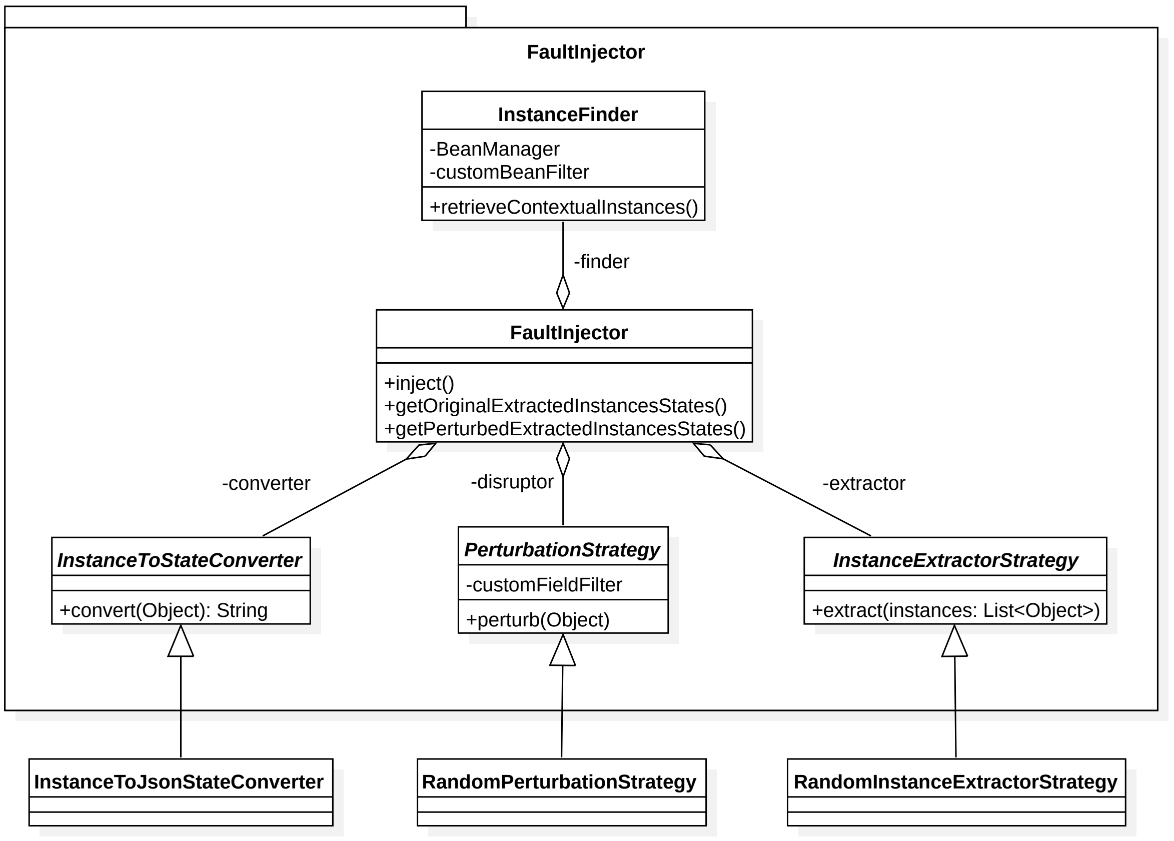

FaultInjector
InstanceFinder
-BeanManager
-customBeanFilter
+retrieveContextualInstances()
-finder
FaultInjector
+inject()
+getOriginalExtractedInstancesStates()
+getPerturbedExtractedInstancesStates()
-converter
-disruptor
-extractor
InstanceToStateConverter
+convert(Object): String
PerturbationStrategy
-customFieldFilter
+perturb(Object)
InstanceExtractorStrategy
+extract(instances: List<Object>)
InstanceToJsonStateConverter
RandomPerturbationStrategy
RandomInstanceExtractorStrategy

Figure 4.3: Class Diagram of the Fault Injector Tool.

`PerturbationStrategy`, `InstanceExtractorStrategy`, and finally `InstanceToStateConverter`.

`InstanceFinder` is used to retrieve the living components in the *business logic* layer of the application under test. The implementation distributed in the repository supports experimentation on applications based on the *CDI* specification and uses the *service provider interface* (SPI). This is the single point of dependency on the specific lifecycle management technology so that `InstanceFinder` is the only class to change to port the fault injector to operate with other frameworks (*e.g., Spring*). The attribute (`customBeanFilter`) allows the selection of IUT components that must be excluded from the effects of fault injection.

Once the component instances are retrieved, the fault injector uses the class `InstanceExtractorStrategy` to extract a component (or a group of), this class has to be implemented in order to provide the extraction strategy, note that this is the point to infer different extraction probabilities for each state component that could be also dependent on the current state composition; in the repository, a random extractor strategy is provided through the `RandomInstanceExtractor` class.

When a component (or a group of) is extracted from the current active instances group, the fault injector perturbs it through a specific strategy encapsulated in the `PerturbationStrategy` class. With the implementation of this class is possible to specify a particular type of perturbation even field-specific[1] and so, it is possible to emulate the different type of problems like DB malfunctions, IoT failures, signal jamming or gamma rays; the repository also provides a concrete implementation (`RandomPerturbationStrategy` class) that performs the perturbation of an individual component field through the *Java Reflection API*. Also note that similarly to the finder class, it is possible to specify a custom field filter allowing the preservation of particular field types from the perturbation.

Finally, after the injection, the fault injector offers the possibility to retrieve a representation of the target component (or the group of) before and after the perturbation through the conversion implemented by the `InstanceToStateConverter` class, in the repository is provided a specific implementation that converts the state of an application component in the JSON format (`InstanceToJsonStateConverter` class).

Orchestrator

The Orchestrator is implemented following the structure of a testing class: the *suite* of usage *scenarios* of the IUT is taken as input and converted in a test suite so that each usage scenario $\{\langle pag_n : act_n \rangle\}_{n=0}^{N}$ (see Sect. 4.2.1) corresponds to a test case allowing its controlled execution.

During each test case, the driver, the third-party tool named Arquillian Drone which extends the *Selenium Web Driver* Gojare et al. (2015), is used to execute the underlying execution scenario $\{\langle pag_n : act_n \rangle\}_{n=0}^{N}$ by performing for each step i the action act_i on page pag_i. However, the Orchestrator does not simply deal with the scenario execution rather, it is also responsible for the activation at the right time of the Fault Injector and monitoring of the IUT parameters. To this end, the Orchestrator requires the implementation of the boolean function `shouldInject()` which is used to guide the fault injection activation before performing action act_i of step i; once landed on page pag_{i+1} through action act_i, the Orchestrator checks if any top-level failure is occurred (by comparing expected with actual displayed values).

Usually, test suites with driver-like capabilities do not allow access to the actual component instances living in the back end, preventing the activation of the fault injector in combination with the user interface interactions and inspections. To overcome this limitation, the *Arquillian* Ament (2013) ecosystem has been adopted in conjunction with the *Arquillian Warp* extension for accessing all the contextual instances living server-side before, or after, a simulated user interaction.

Finally, for each executed scenario, different indicators are collected:

[1]For example, if the field extracted is an *ordered list*, it could be defined a type of perturbation that changes the order of the list items, alternatively, if the field is a *string* that requires a specific input format (e.g., a mail address) it could be defined a perturbation that changes the address with another random one, still respecting the constraint (e.g., email regexCrocker et al. (1982)).

- *the final state of the application*: it is interpreted as the collection of the living contextual instances after the completion of the last HTTP request of the scenario;
- *the state of the component*: during the execution, the fault injection is triggered at a time defined by the `shouldInject()` function perturbing one of the living components. Due to the implemented rejuvenation strategy and the error propagation mechanism, it may be difficult to retrieve the original perturbed component simply by inspecting the final application state. To this end, after the injection, the component's fully qualified name, its scope and the type of perturbation are saved;
- *the number of failures manifested during the simulation*: in an FEF chain perspective, only top-level failures (i.e., failures manifested on the UI and visible to the end-user) have been considered, thus neglecting failures of intermediate components.

Let us consider the following scenario execution of the IUT:

$$\begin{aligned}
&\langle HomePage, clickLoginButton()\rangle\\
\rightarrow\ &\langle LoginPage, performLogin()\rangle\\
\rightarrow\ &\langle MeanFirstPage, insertFirstOperator(\text{"10.0"})\rangle\\
\rightarrow\ &\langle MeanSecondPage, insertSecondOperator(\text{"20.0"})\rangle\\
\rightarrow\ &\langle ConfirmationPage, clickReturn()\rangle
\end{aligned}$$

The Orchestrator tool will convert it in a test case made of 5 steps, which template illustrating the implementation style is reported in Listing 4.1.

The `Warp.initiate()` method (*line* 5) defines what happens in a single user interaction from the end-user perspective, defined in turn by the `Activity` class definition through the overriding of the `perform()` method (*line* 7). Assertions and behaviors from the server-side perspective are defined in the `inspect()` method (*line* 15) through the `InjectionInspection` class definition; since considered scenarios are composed by a sequence of user interactions, their implementation will be a sequence of invocations of `initiate()` and `inspect()` methods. The snippet also shows the invocation of the prescribed user action (in the particular case act_2 i.e., `clickLoginButton()` at *line* 13) only after the evaluation of the occurrence of a top-level failure (i.e., `checkPageCorrectState()` at *line* 10) within the current page (in the particular case pag_2 i.e., `loginPage`). The `InjectionInspection` class has been defined for setting up some action hooks, through *ad hoc* annotation decorators

```java
@Test
public void testScenario1() {
 ...
  // A user interaction
  Warp.initiate(new Activity() {
    @Override
    public void perform() {
      loginPage = new SimpleAppLogin(driver);

      if (!loginPage.checkPageCorrectState())
        failureOccurred(runInfoFilePathStr);

      login.performLogin();
    }
  }).inspect(new InjectionInspection() {

    String path = testDirStr;

    boolean executeFaultInj = shouldInject();

    @Inject
    BeanManager bm;

    @BeforeServlet
    public void injectFault(){
     if (e  xecuteFaultInj)
       injectFault(bm,
             path + "/errorInjected");
    }

    // Invoked only in the final interaction
    @AfterServlet
    public void saveFinalState() {
     printState(
         instanceFinder.
            retrieveContextualInstances());
    }
  });
 ...
```

Listing 4.1: Template of a navigation step implemented by the Orchestrator.

```
 1 @Deployment(testable = true)
 2 public static WebArchive createDeployment() {
 3   WebArchive war = ShrinkWrap
 4   .create(WebArchive.class, "deployment.war")
 5   .addPackages(true, "appPackageName")
 6   .addClass(InjectionInspection.class);
 7   .addClass(FaultInjector.class)
 8   .addClass(InstanceFinder.class)
 9   .addClass(InstanceToStateConverter.class)
10   .addClass(PerturbationStrategy.class)
11   .addClass(InstanceExtractorStrategy.class)
12 
13   return war;
14 }
```

Listing 4.2: Custom deployment for scenarios execution.

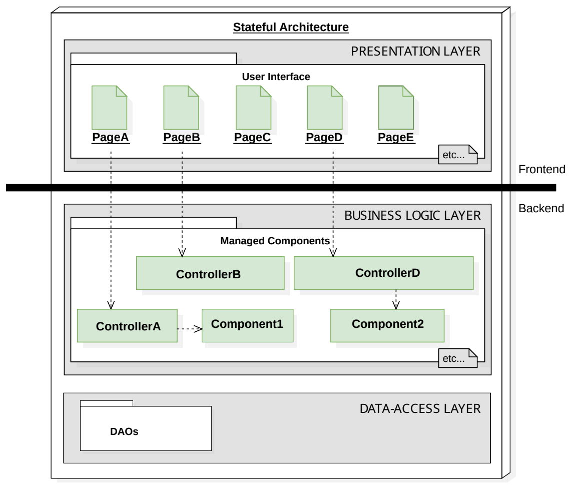
(i.e., `@BeforeServlet` and `@AfterServlet`), where invoke specific methods (e.g., `injectFault()` and `saveFinalState()`). Specifically, the `injectFault()` method (*line* 25) is responsible for triggering the fault injection if required (i.e., `shouldInject()` evaluation at *line* 19), and save the state of the perturbed component. Finally, `saveFinalState()` method at *line* 33 retrieves all the component instances in order to save them within a file in the JSON format.

Note that, through the *Arquillian* deployment mechanism, which adopts *ShrinkWrap* Hat (2016) for the creation of Java archives files, our implemented module does not need to be integrated within the production source code of the IUT, thus enabling an experimentation stage with "no modifications" on the Web Application under test.

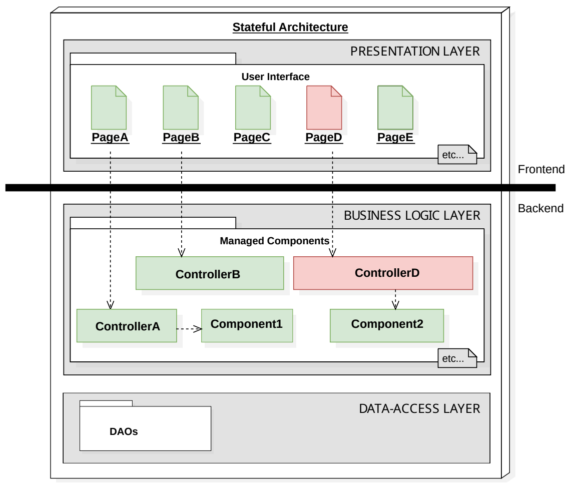
Listing 4.2 represents the deployment of the archive referred to the application (i.e., usually a *.war* file) that is essential for the configuration of the test environment. This is accomplished by implementing the *public static method* annotated with `@Deployment` that returns the archive. As mentioned above, the approach allows dynamic insertion of classes into an application without modifying the real source code of the IUT; this also enables the design of tailored archives with only the classes needed for the test suite, thus obtaining lighter deployments and, consequently, speeding up test executions. As appearing at *line* 5, it is possible to add entire packages to the deployment (in this snippet, the root package of the IUT is added), but also to add single classes, as in *lines* $6 - 11$ where classes for the classes of fault injector tool described previously are added.

4.2.3 Observed effects of fault injection

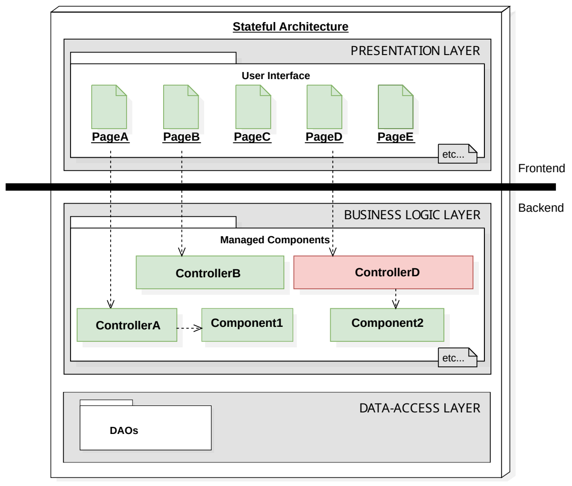
To correctly study the effects of errors in the IUT, different behavior has to be taken into account: a fault activation does not necessarily causes an immediate failure,



(a) Error corrected scenario



(b) Manifested failure scenario



(c) Latent error scenario

Figure 4.4: Possible states considered after each execution during experimentations

it could rather lead to an erroneous internal condition (i.e., the erroneous state) remaining silent for an unpredictable period of time or evolving into further errors (i.e., error propagation). Under these assumptions, a failure could be caused by a set of events occurring over a long-term period of time, making the failure detection and the subsequent fault removal phase extremely difficult Grottke and Trivedi (2007).

For all these reasons, not only failures caused by the fault injection are considered but also the possible configuration of the IUT internal state (also represented graphically in Fig. 4.4); to do this, it was used the implementation described in Sect. 4.2.2 in particular, exploiting the raw data given by the indicators collected by the Orchestrator and comparing the final state of the application after each scenario execution with the state of a "clean" run: execution of the same scenario which, however, is not subject to fault injection, thus acting as a ground truth. Doing so, each scenario execution is classified into one of the following categories:

- **manifested failures**: top-level failures manifested during the execution (Fig. 4.4b); a manifested failure highlights that errors have deviated from some external states of the system also affecting functionalities and services offered through the UI, thus implying that the rejuvenation policy was "too soft". This information is derived, addressing the number of failures observed by the Orchestrator during each execution;

- **latent errors**: errors that do not contribute to top-level failure manifestations either are corrected by the rejuvenation policy (Fig.4.4c); a latent error remains hidden even after the end of the execution, since a usage scenario represents only a possible navigation path, and further interactions are not taken into account, the absence of failures during the execution does not guarantee that it will not eventually manifest. Thus, this outcome describes a situation where the state of the application is corrupted by software aging and the rejuvenation policy has not been yet applied. This information is derived by comparing the final state of the "clean" run with the final state of the examined execution (if at least one component state differs from its counterpart obtained after a "clean" run and there are no manifested failures in the execution, then a latent error occurs);

- **corrected errors**: errors that are automatically corrected by the Dependency Injection container (Fig. 4.4a); the correction of the errors outlines the success of the rejuvenation strategy defined through the designed component scope. This information is derived by comparing the final state obtained after a "clean" run with the final state of the examined execution (if there are no differences and if no failures have occurred, then the error has been corrected successfully).

Besides, during the experimentation, information about **errors propagation** was collected. (i.e., errors that manifest and propagate failures in external components dependent on directly affected ones, also producing the activation of external faults and consequent errors) that increase the number of possible sources of failures. Specifically, the observation only considered propagations affecting dependent components that are in an erroneous state at the end of the scenario execution (neglecting cases in which the propagation is corrected by a rejuvenation of the component). This information is derived by comparing the final state of the "clean" run with the final state of the "faulty" run (i.e., if other components near the one, affected by the injection, are in an erroneous state - different from the state obtained after a "clean" run - then the error propagation occurs).

Errors propagation is a crucial parameter for the overall sensitivity to errors: it gives rise to the error accumulation problem Dohi et al. (2020) (Fig. 4.5), complicating the correction (i.e., more than one component has to activate the rejuvenation strategy to fix the error, also generating "chain reactions" where more components could cause failures and propagate errors, *recursively*).

Although these measures give an idea of the actual error propagation for each scope, they only partially express how components could spread errors throughout the IUT and to complete this observation, the propagation capacity was also monitored for each scope. In order to assess this measure, a "dirty bit" is added on each instance which indicates whether a component has been exposed to the possibility of being contaminated with an erroneous state by another instance, during the run, every time a component, with the dirty bit set to true, interacts with another component the dirty bit of the latter becomes true. Note that, the component interaction involves both sides of the dependency between scoped components: a dirty component (with the dirty bit enabled) could use a method of a component with the dirty bit disabled and then dirt it, on the other side, a clean component (with the dirty bit disabled) could call a method of a dirty component and, consequently, getting itself dirty.

4.3 Experimentation

Experimentation was carried out in two different flavors: a *scope-wise experimentation* (Sect. 4.3.1) aimed at measuring how components with different life cycle scopes impact the overall IUT unreliability; and a *policy-wise experimentation* (Sect. 4.3.2) aimed at evaluating how different strategies of life cycle design can protect the IUT.

Both the experimentations were executed on a simple JEE (*Java/Jakarta Enterprise Edition*) Web Application, developed so as to exemplify the mechanisms of interaction occurring between the User Interface and the Business Logic of a 3-tier stateful architecture, with a User Interface made of 6 main pages, and a Business Logic made of 8 components, 5 acting as Controllers serving page requests and 3 additional Beans, using CDI (*Context & Dependency Injection*) as Automatic Lifecycle Manager. As mentioned in Sect. 4.2, the tools performing fault injection and state monitoring do not require access to the source code of the application under test. However, different variants of the same application were used, with the same functionality and pages featured in the User Interface but different specifications of components scopes.

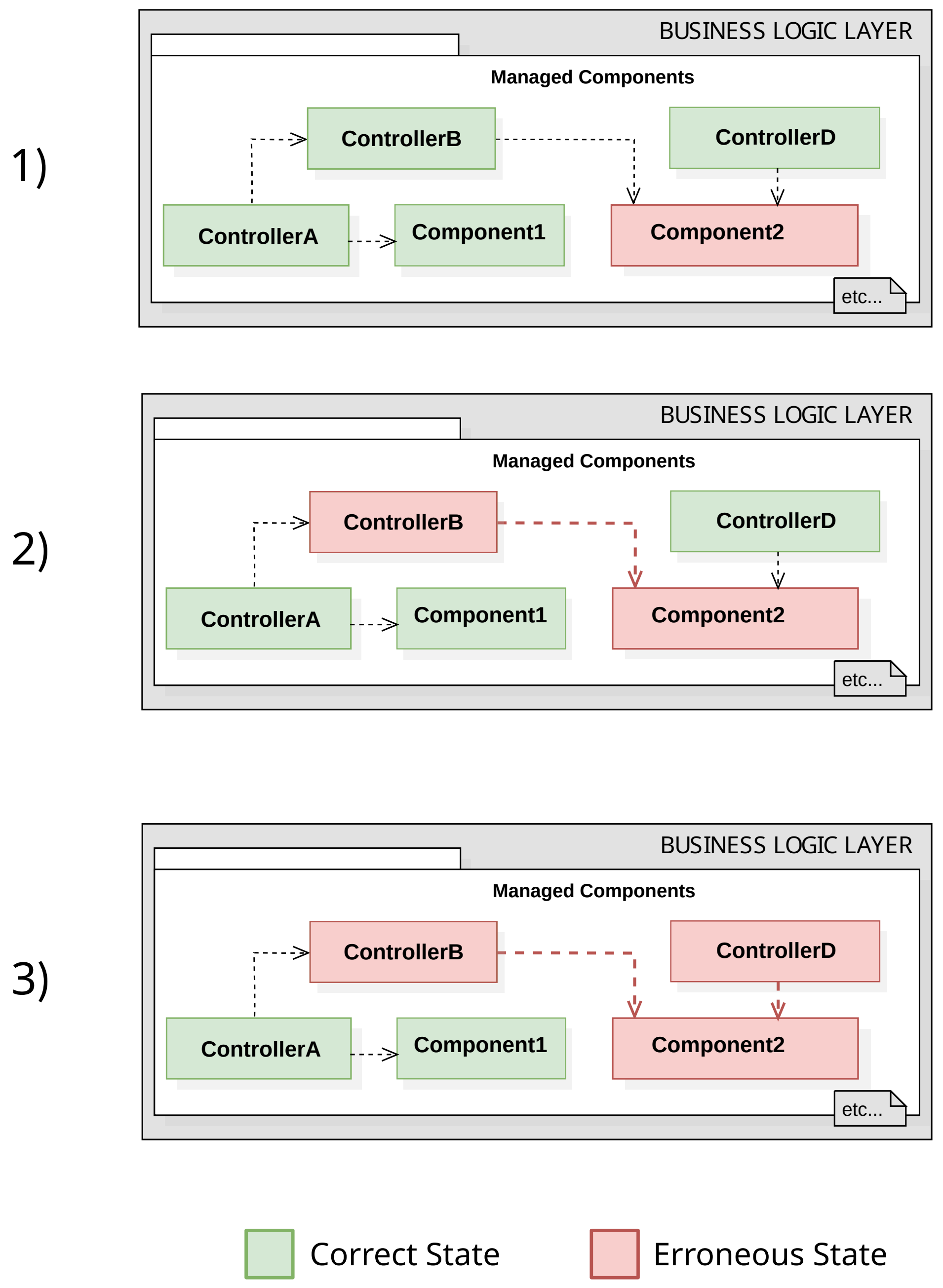


Figure 4.5: Error propagation phenomenon among application logic instances. In *1*) component called `Component2` is in an erroneous state, in *2*) error is propagated through the interaction from `Component2` to `ControllerB`, finally in *3*), the situation deteriorates further with another error propagation affecting `ControllerD`.

Table 4.1: Scope-wise experimentation results.

Scope	**Manifested failures** (%)	**Latent errors** (%)	**Corrected errors** (%)
application	42.4	57.6	0
session	48	36	16
conversation	12	24	64

4.3.1 Scope-wise experimentation

The *scope-wise experimentation* aims at observing how components with different life-cycle lengths are hit by errors and how these errors are then propagated or corrected by the component refresh. To this end, the experimentation was performed with a variant of the IUT with a balanced use of scopes: 2 components with application scope, 2 with session scope, 1 with conversation scope and 3 have request scope. In the experiments, 2 different scenarios were considered with different lengths of 5 and 10 user interactions, respectively, and their execution was repeated 100 times each to randomize sojourn time in visited pages and the arrival time of injected faults. In each experiment, a single fault was injected.

Results are outlined in Tab. 4.1, reporting, for each scope of the component hit by the fault injection, the fractions of errors that are: corrected before the end of the scenario by a refresh of all affected components states (Corrected); or propagated up to appear as a failure in the functional behavior exposed by the User Interface before the end of the scenario; or propagated until the final state at the end of the scenario, but not manifested as a failure (Latent).

By design, in the *application* scope, the state is never refreshed, i.e. no rejuvenation is applied, and no errors are thus corrected; besides, the *conversation* scope corrects more errors than the *session* scope, as the component is refreshed after a shorter activity cycle. Results for the *request* scope are not reported, as they are continuously refreshed, at each HTTP interaction, so that the probability that the fault injection hits a specific life interval and this can even be propagated to other components is negligible with respect to what observed for the other scopes.

It is also interesting to examine how components associated with wider scopes are more likely to be picked up by the fault injector (the experimentation has shown a sampling probability of 0.625, 0.25 and 0.125 for *application*, *session* and *conversation*, respectively). This intrinsically reflects the stochastic characterization of the approach, adopted in sampling times and choosing components for the fault injection mechanism, described in Sect. 4.2.1: components with wider scope live longer and thus have a higher probability to be living at the time of fault injection.

Table 4.2: Error propagation in scope-wise experiments.

Scope	**Errors propagation** (%)	**Mean propagated errors**	**Mean touched components**	**Prop. ratio** (%)
application	44	1	4.6	21.7
session	32	1.33	2.5	53.2
conversation	16	1.25	1.7	73.5

Tab. 4.2, reports measurements related to propagation in the same scope-wise experiments, according to metrics introduced in Sect. 4.2.3. Results are grouped according to the scope of the component first perturbed by the fault injection, and they report in the first two columns the fraction of executions where the error was propagated by at least one hop (*Errors propagation*); the mean number of components touched by the error propagation (excluding the initially perturbed component from the count) with an erroneous state at the end of each simulation (*Mean propagated errors*).

Note that the *application* scope has, on average, a lower number of propagated errors, but the propagation occurs more frequently with *application* scoped components (44% of the times against the 16% of the *conversation* scoped ones) and in addition, the mean touched components for each scope are directly proportional with the lifecycle of the scope.

The last two columns of Tab. 4.2 also reports: the ratio between the "Mean propagated errors" (*Prop. ratio*) and "Mean touched components", which counts how many components have been effectively affected in average w.r.t. the components that might have been affected. This provides an interesting insight: *in absolute*, wider scoped components result in propagating much more errors, but narrower scoped components have a higher rate of deviating the runtime behavior of interacting components (i.e., activating some external faults in dependent components). This is probably due to the fact that implementations of narrower scoped components are strictly related to the inner business logic of use cases, thus generating stronger relationships and implementation couplings between components. Thus, the propagation ratio seems to suggest that not only the component scope affect the reliability but also the interactions of the components act an important role in the propagation phenomenon.

4.3.2 Policy-wise experimentation

The *policy-wise experimentation* aims at measuring how a general policy in life cycle design can influence the immunity of the IUT to errors. To this end, this second

Table 4.3: Policy-wise experimentation results.

Principle	**Manifested failures** (%)	**Latent errors** (%)	**Corrected errors** (%)
data long retention	42	46	12
lower scope	30	26	48

experimentation was carried out evaluating the effects of the same injected faults on two *functionally equivalent* versions of the IUT designed under distinct principles for the lifecycle design of managed components:

- the *data long retention principle* promotes the use of wide scopes (e.g., *application*, *session*), allowing in-memory information retention for longer time intervals so as to reduce the need to repeat retrieval or evaluation of the same data; this increases responsiveness and reduced the energy footprint, but, as a drawback, it also increases the memory load and the space for error propagation;
- the *lower scope principle* promotes the usage of scopes as narrow as possible (the best case being *request*), minimizing memory occupation and exposition to aging effects, but requiring a dedicated place to store runtime data (e.g., the local or session storages in the client-side or dedicated databases in the server-side), thus producing an overhead in computation.

The same 2 scenarios were considered as for the scope-wise experimentation, and their execution was repeated 50 times for each variant of the IUT. Also in this case, in each experiment, a single fault was injected.

Results of the *policy-wise experimentation* are summarized in Tab. 4.3; Values are consistent with the *scope-wise experimentation*: since under the *lower scope principle* is plausible to assume that components are predominantly *conversation* or even *request* scoped, while under the *data long retention principle* the scopes will be mainly *session* and *application*.

While providing higher immunity to errors, the *lower scope principle* cannot yet be considered a silver bullet for every case; indeed, the selection of a specific policy implies a trade-off between the ability to correct errors Cotroneo et al. (2013) and the related computation overhead.

If the rejuvenation is rarely applied, errors could be activated and propagated, conversely, if the rejuvenation is frequently applied, the components state needs to be stored in other dedicated places (e.g., the DB) increasing the number of operations required to manage it (e.g., DB transactions).

Chapter 5

Towards Runtime Verification of Application Logic

In the previous Chapter, it was demonstrated that in software architectures with a session state subject to software aging, the automated life cycle management can relieve the phenomenon by acting as a mechanism of software micro-rejuvenation. In particular, experimental results show that components with a narrow life cycle are less prone to manifest top-level failures than those with an extended life cycle and, at the same time, the error propagation measures, seem to suggest that components interactions act a central role for the propagation among multiple instances, thus making component relations a feature as critical as the life cycle length for the overall reliability of the architecture. However, generalized adoption of minimal scopes for all components, as in the case of the lower-scope design policy, does not represent a viable solution: business transactions continue to require the management of a session state (Sect. 2.2) that, if not retained server-side (as in stateful architectures) or client-side (as in service-oriented architectures), must be kept somewhere else, usually in the database, with various drawbacks like performance degradation (Fowler (2012)) as its management requires the use of transactions even for transient and temporaries information which usually are subject to frequent manipulations.

On the other hand, designing the application logic with the aim of minimizing the dependencies among components may lead to code duplication, poor maintainability and testability (Sect. 2.2), besides the fact that this practice could result in an overshoot: the number of interactions between components instances occurring dynamically during their life cycle may be much less than the statically declared dependencies of the corresponding classes.

In order to observe this dynamic behavior of the session state and represent both the lifetime and the entertained interactions of each component instance during a usage scenario, it is proposed an abstraction identified as *timeline*.

On top of that with the aim of identifying a micro-rejuvenation policy maximizing the reliability of the system without jeopardizing performances, it is proposed a solution in the field of *runtime verification* Bartocci et al. (2018) based on source code level software instrumentation which by the way, was also developed concretely as an adaptable *plug-and-play* tool for stateful JEE architectures.

This chapter is organized as follows: in Sect. 5.1 it is presented a timeline abstraction capturing both the lifecycle and the internal interactions of managed components and in Sect. 5.2 it is proposed a solution based on a classic runtime verification configuration aimed to exploit the expressivity of the abstraction to implement a tailored and component-wise software rejuvenation strategy.

5.1 Making components interactions explicit

Dynamic interactions of component instances and evolution of the session state are regulated statically during the application logic implementation (i.e., component class definition and dependencies specification), however, the actual lifetime of a component instance and the number of interactions performed with other instances are guided by the sequence of user interactions carried out by the client using the application. This mechanism makes it difficult for a developer to figure out how the application logic will behave at run time, with this purpose it is introduced the *timeline* abstraction.

An example of this timeline abstraction is depicted in Fig. 5.1: the *x-axis* marks the succession of inputs received from the user interface (requests) along continuous time, for the sake of conciseness irregular occurrence of requests in time is represented with a timestamp associated with each request which is represented as instantaneous epochs preserving the qualitative order of interactions among time (equivalently to the application logic representation of Fig. 2.4 represented in Sect. 2.4). The *y-axis* instead, represents the components types that live among multiple requests during which they can interact with each other. Concretely, in request r_3, which arrives at time t_3 identified as the timeshift of δ_3 with respect to the previous t_2, a component instance of type c_3 is created and it is subsequently destroyed at request r_5, during its life as represented by the link at request r_5, it interacts with a component of type c_2 which in turn interacts with components of types c_1 and c_4.

Propagation of errors in components states due to data-flow def and use actions Rapps and Weyuker (1985) are captured by vertical links, which thus represent a way for a possible error to be propagated from one component to another, no matter what the directionality expressed by the relation.

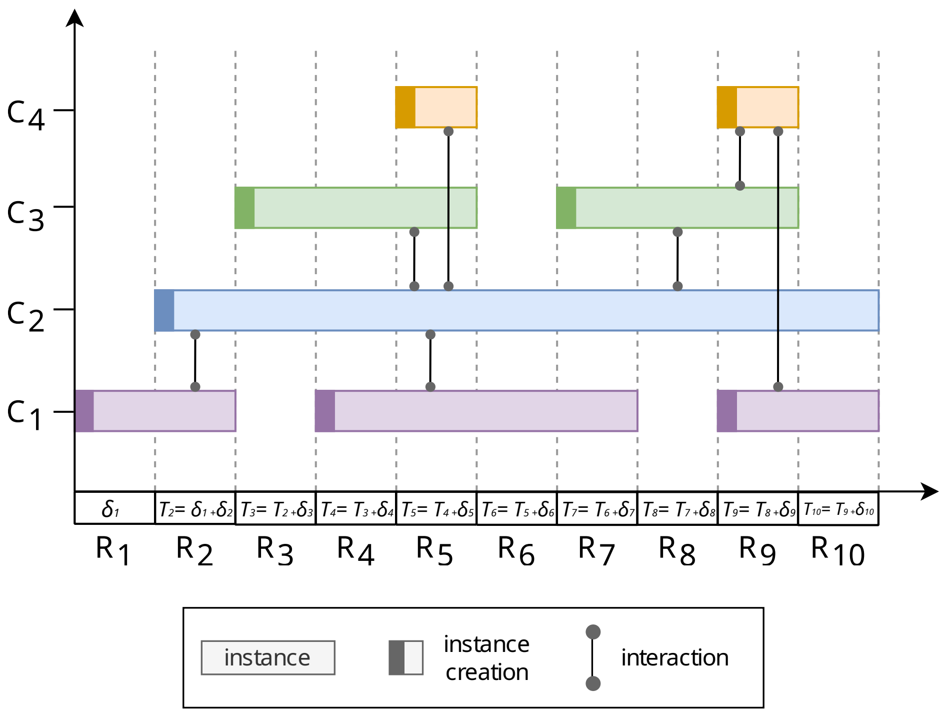


Figure 5.1: Timeline abstraction of overlapping cycles and dataflow dependencies among 4 component types along a user interaction with 10 steps.

5.1.1 The Propagation Range

The timeline abstraction is a good tool to study and understand how an error could spread among component instances, to better depict the phenomenon, it is introduced here the concept of *propagation range* identifying which component instance might be affected by a specific error activation.

The propagation range group $PropRange_{r_b,r_e}(c)$ is defined as the group of components instances that, during the ordered sequence of requests starting at request r_b and ending at request r_e, have had the opportunity to influence each other and consequently to spread an error activated at request r_b in the state of component c. Initially, c is the only member of the group, however as the requests come in, the number of alive members may vary: some components are added as they interact with a member of the epoch, and some others die because they have completed their life cycle; when no more members are alive it means that the error can not be survived to the rejuvenation policy.

As an example, consider again the timeline of Fig. 5.1, if an error activates at request r_1 in component c_1, $propRange_{r_1,r_1}(c_1)$ will contain only c_1, however, $propRange_{r_1,r_{10}}(c_1)$ will contain all the component instances in the represented scenario meaning that instances c_1 and c_2 at request r_{10} may be affected by an error propagated by their contemporaries or by dead instances and originated at request r_1.

The propagation range brings important information about the state of the application:

- the length of the underlying requests sequence says how long an error could be latent among the members;
- the number of alive members at a certain time says how wide the error propagation could be in the worst scenario.
- it identifies a cluster of components in which, the error has no possibility of being propagated to non-members components.

5.2 Combining Rejuvenation with Runtime Verification

It is clear that the application logic design identifies a number of data flow coupling scenarios which is unfeasible to consider beforehand at implementation time: the application could hide frequent unexpected and sometimes counter-intuitive patterns (e.g., a lower scoped component that often lives for an extended period of time) or conversely some rare input sequences could bring the system to a failure with high probability.

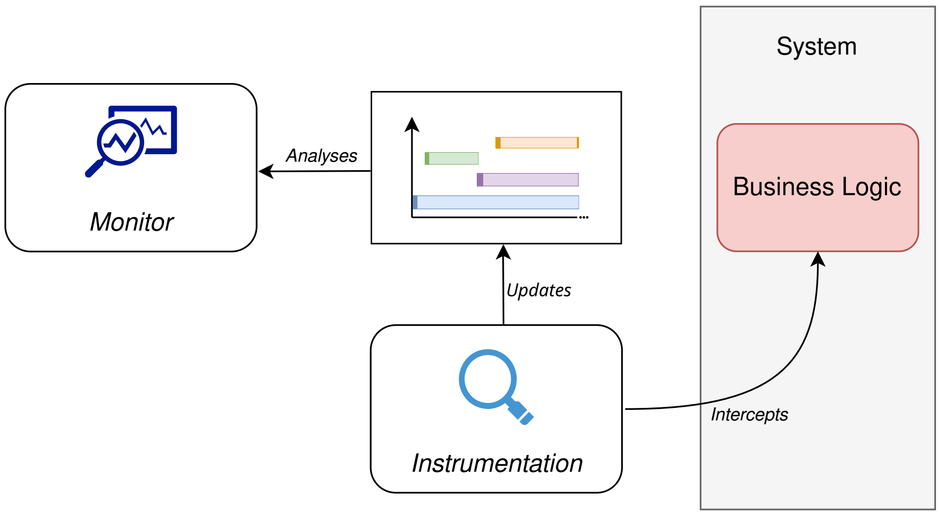


Figure 5.2: The runtime verification setup.

For these reasons, it is proposed a solution aimed to exploit the timeline abstraction beyond the simple graphical support use, to do this, it is proposed a monitor setup that, as can be seen in Fig. 5.2, is heavily inspired by configurations often proposed in the runtime verification field and consists in: the *system*, an *instrumentation* that runs concurrently to the system observing it and finally, a *monitor* instance that processes the observation.

Each request made by the client on the user interface triggers the instrumentation which in turn records the application logic behavior during the answering process of the system, in particular, it collects components interactions, instantiation and destruction and adds this information to the execution trace as a system event for the monitor. Note that, a system event takes the form of a request epoch and in turn, the execution trace takes the form of a timeline progressively built while the system is used.

With this workflow, the system trace could be processed with an offline monitoring activity, in this case, the obtained timeline results in a powerful tool to analyze how and when the session state enters a configuration particularly prone to failure manifestation and then change the scope configuration of specific components according to it. It could be used to perform predictive analysis (*what-if analysis*) on a usage scenario: what could happen if at time t during request r the component c enters in an erroneous state? To this end, the propagation range defined previously could be used to assess the worst-case scenario. However, if done with the aim of improving the scope design of the application, the predictive analysis could be dispersive due to the endless number of combinations and scenarios to consider, instead, it could be carried out a reliability evaluation for each component.

For this purpose, referring to the experimental results obtained in the previous chapter, the timeline abstraction offers all the information needed to assess component-wise reliability: a component instance can be subject to aging depending on its intrinsic sensitivity and its current lifetime, at the same time, a component could be entered an erroneous state also during an interaction with another component with a probability that depends on the reliability of the latter. Thus, the proposed reliability metric depends not only on the actual life span but also on the level of reliability of the components with which the component of interest has interacted so far.

As a showcase let us consider one more time the abstraction of Fig.5.1 and let us assume that this timeline was generated by an actual system, deepening its composition, it can be seen that it captures a scenario made of a sequence of 10 subsequent requests $\{r_1, r_2, r_3, ...\}$ involving 4 component types $\{c_1, c_2, c_3, c_4\}$ with different life cycles i.e., instances of component c_1 live within the sub-sequences $\{r_1, r_2\}$, $\{r_4, r_5, r_6, r_7\}$ and $\{r_9, r_{10}\}$, c_2 instance lives from request r_2 to the end of the scenario etc…

The representation outlines the presence of a specific instance, c_2, that according to the above described metric, shows a high level of unreliability: assuming high values of δ, at request r_{10} has accumulated a considerable amount of uptime in addition to 5 interactions with other components, this fragility is also confirmed by the propagation range $propRange_{r_1,r_{10}}(c_1)$ discussed above and enforced by the propagation scenario depicted in Fig. 5.3a.

Note now that changing the scope configuration adopting a drastic design policy like *lower scope* (described in Sect. 4.3), will relieve the situation: as can be seen in Fig. 5.3b the error can be propagated only to the instance of type c_2 and the propagation range stops at request r_2 thanks to component c_2 that now ends its life cycle early. However, this solution comes with a high price: the state of various components previously kept in memory among multiple requests and not particularly threatening for reliability, now have to be maintained somewhere else, probably stored in the DB, requiring several additional transactions to implement the same scenario.

Relying on the timeline instead, it is possible to elaborate a less conservative, but still effective, tailored solution: just lowering the life cycle of c_2, the component identified by the reliability metric as the weak point, will cause a significant reduction of the error propagation potential with a minimal impact regarding the number of additional transactions as represented in Fig. 5.3c.

The above mentioned strategy is configured as an offline monitoring analysis that can be conducted after a phase of a *in-vivo* scenarios collection, however, the instrumentation and the reliability metric could be exploited also to implement a *synchronous* online monitoring process, in this case, the timeline changes its usefulness moving from being an analytical tool able to provide insights of the system, to be a heuristic for the implementation of adaptive micro-rejuvenation strategy.

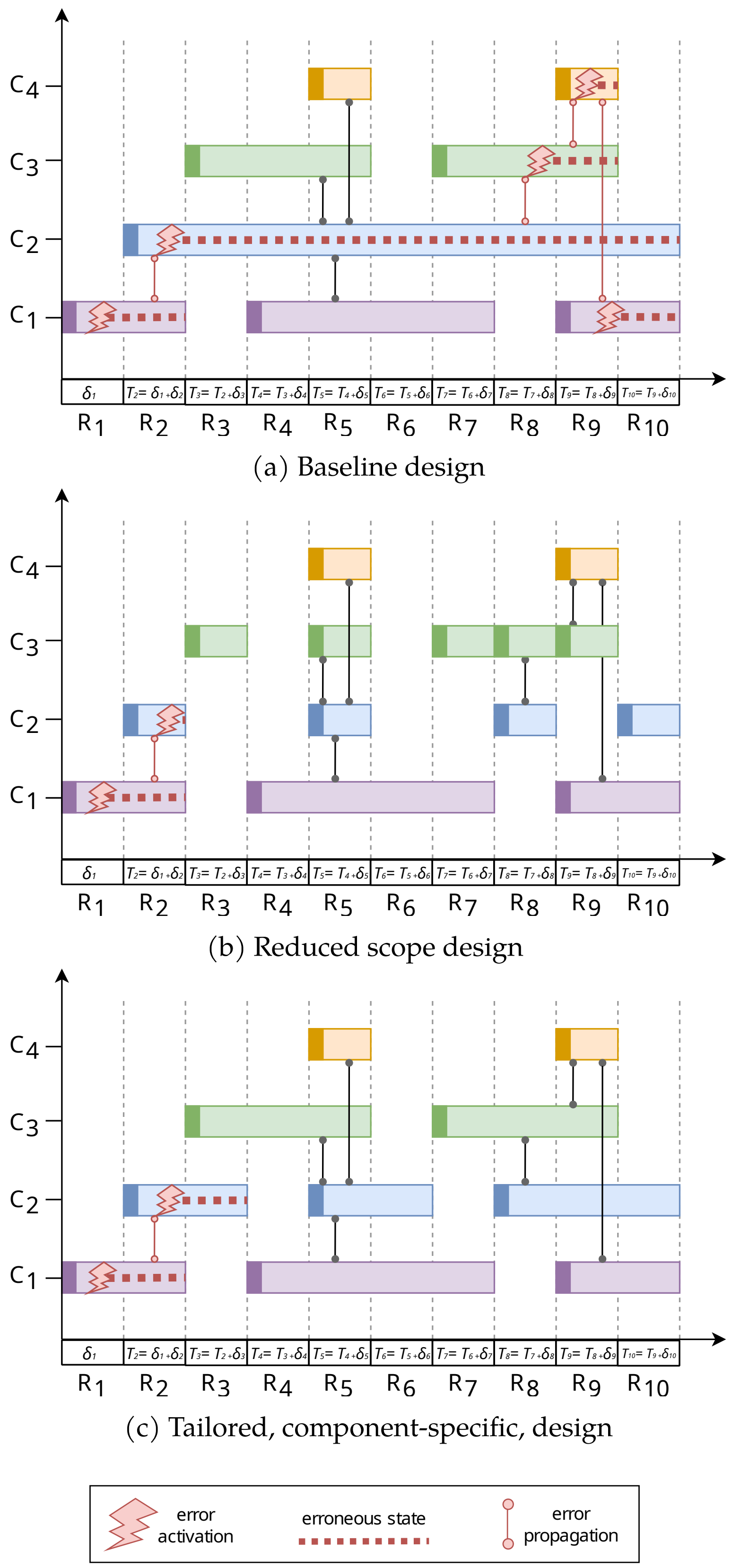
5.3 Implementing Instrumentation of a Runtime Verification Environment

To experiment with the proposed approach, an instance of the instrumentation represented in Fig 5.2 has been developed in the form of a software tool named *beanInterceptor*, able to inspect Stateful JEE architectures.

The tool is developed relying on several advanced mechanisms of JEE: a CDI extension has been developed combining also CDI interceptors, listeners and the Service Provider Interface of CDI.

It can intercept each request received and distinguish among multiple parallel sessions:

- the arrival time;
- the set of instances created during the response process;
- the sequence of methods called (the interactions);
- the set of instances destroyed at the end of the response process.



(a) Baseline design

(b) Reduced scope design

(c) Tailored, component-specific, design

Figure 5.3: Comparison of overlapping cycles and dataflow dependencies under different policies of scope design.

Thanks to its implementation, *beanInterceptor*, does not require to modify the code of the target system and it can start its job by simply deploying it in the same application server of the system and automatically it will gather all the information to build the timeline, however, it is also possible to fine-tune its behavior defining ad-hoc filters to ignore some components and identify the core application logic.

This implementation enabled the study (currently offline) of the application logic of a real-world application specifically an Electronic Health Record in use for several years in a major Hospital of Tuscany Region Patara and Vicario (2014)Fioravanti et al. (2016) made of $\approx$ 35 pages, $\approx$ 30 DAOs and $\approx$ 35 domain classes supported by a wide internal library of $\approx$ 200 classes.

Chapter 6

Conclusion

This final chapter summarises the contribution of this dissertation and then describes works in progress and proposes possible further steps.

6.1 The Contribution of this Dissertation

This dissertation contributes to a characterization of how crucial the development of the business/application logic is for the overall reliability of the software architecture. Specifically, this document identifies two major threats in the configuration of the business logic both related to instances couplings that occur dynamically during the usage of the software system, and, for each of them, it provides a strategy to decrease failures occurrence.

In case of faults hidden in the components implementation, it was provided a model based methodology that reinterprets the data flow testing practice at the architectural level (gray-box) and which is aimed to identify and execute usage scenarios that could identify the faults, basing its selection on the business logic composition and the DI container behavior.

In presence of external faults affecting the state of the system, as for the case of software aging, experimentation was first conducted to verify if the automated life cycle management implemented by DI frameworks, can somehow address these events, once demonstrated that the DI container acts a kind of micro-rejuvenation procedure on the managed components, an adaptive procedure based on the dynamic state of the system has been proposed combining practices of software aging with practices of runtime verification.

6.2 Future Works Open

The presented dissertation opens the way to new research works in various directions.

Ongoing work is focused on the application of the runtime verification setup to the real-world application discussed, with particular effort on implementing a synchronous online monitor making it possible to refresh components dynamically and identifying heuristics based on the observed timeline.

These directions allow several additional studies, in particular, the trade-off between the reliability and performance of the system should be characterized formally to identify optimal strategies for rejuvenation.

Another research line of interest consists in using these techniques and knowledge, especially the one related to runtime verification, in the field of cyber-physical systems and Internet of Things (IoT) architectures where external faults and aging are frequent problems.

Appendix A

Additional Material

A.1 The FlightManager Repository

This Section reports additional materials, providing guidance on access to the accompanying repository and illustrating an excerpt of its contents.

A.1.1 Flight Manager Repository Structure

Flight Manager is a full-fledged stateful web application in the functional context of an online flight booking system; the whole source code and detailed documentation, are made available in a repository hosted on *Github*[1]; this subsection provides a description of the repository structure and explains how to consult the documentation.

The `README.md` file is the starting point for the *Flight Manager* documentation, a brief introduction is provided along with the most standard artifacts, most of which are also presented within this paper. In addition to this, for each use case, a specific documentation page is available with the related *Enriched Robustness Diagram* and *managed component Data Flow Graph*; for instance, the ERD of Fig. A.1 and the mcDFG of Fig. A.2 can be found on the *Search Flight* use case documentation page.

On the same page, we also make available groups of mcDFG paths each of them fulfilling one of the proposed coverage criteria; for example, the screenshot in Fig. A.3 represents these collections reported in the *Search Flight* use case documentation page: note that each path is identified through a sequence of numbers corresponding to the node id of the related *mcDFG*.

The identified use cases test suites are grouped into 5 packages, one for each coverage criterion, resulting in the application test suites; Fig. A.4 shows their structure that can be found at the `src/e2e/java` path of the repository.

[1]Repository link: `https://github.com/LeonardoScommegna/unravel-experimentation`

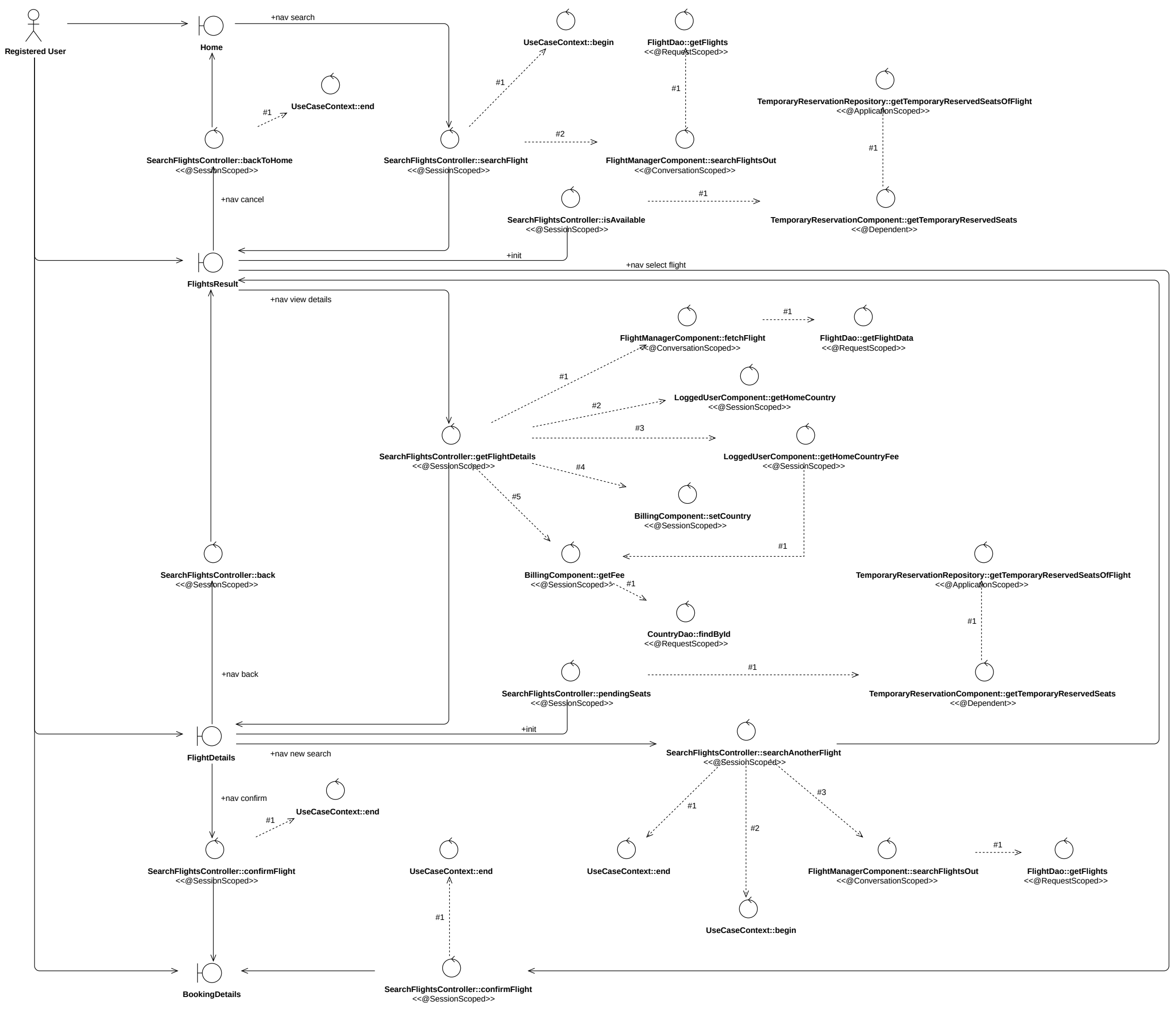


Figure A.1: Enriched Robustness Diagram available at the Search Flight use case documentation page.

The main feature of the *Flight Manager* repository is that it brings the correct version of the web application as well as 32 other versions that differ from the correct one in a manually injected fault. To this end, the repository has $32 + 1$ branches: the first one (the *main* branch) brings the correct version of *Flight Manager* while the remaining ones are faulty versions. The documentation of each faulty version is briefly reported in the `README.md` file where the table (also reported in Fig. A.5) outlines the type of fault, the type of failure induced by the fault, and also the use case involved. However, each faulty version has a dedicated documentation page with additional information, for instance, the diagram of Fig. A.6 can be found in the *Faulty Version* #1 documentation page and provides a graphical representation of the fault through the *ERD* of the *Search Flight* use case decorated with the ✕ symbol representing the fault, that in the specific example consists in the lack of the enclosed context begin. In addition to this, each faulty version documentation page reports information about the fault detection capability of each test suite related to the use case involved, for instance, Fig. A.7 represents the test report for the *Faulty Version* #1; keeping in mind that we consider a test suite capable of detecting a fault if at least one test case fails, thanks to the check marks it is possible to see whether a test suite is successful and also see which particular test fails *i.e.,* all the coverage criteria detect the fault of the example, in addition, the specific case of the *All Edges* test suite, only the first test case detects the fault.

Finally, also the *all pages* and *all navigation* test suites, obtained on top of the page navigation diagram are reported in the documentation in addition also their fault detection capabilities are summarised in a table (see Fig. A.8).

A.1.2 Collection of Issues Related to Automatic Contexts Management

The following collection represents a subset of the available issues encountered by users using frameworks for dependency injection and automatic contexts management. The collection does not aim to contain all existing cases of the problem but to demonstrate the wide range of possible pitfalls that a programmer might face.

- stackoverflow.com: "how end one cdi conversation and completely destroy all variable of CDI bean";
- living-sun.com: "JSF combined with CDI, issue with conversation scope";
- stackoverflow.com: "new-cdi-conversation";
- stackoverflow.com: "CDI conversationscope: end and begin with one request?";
- stackoverflow.com: "Error in JSF + CDI conversation scope when begin is called two times";
- stackoverflow.com: "CDI conversation id is always 1 and NonexistentConversationException caught when trying to resume conversation";
- stackoverflow.com: "Why my spring session scoped bean is shared across sessions?";
- github.com spring-vaadin repository, Issue: "Session scoped component #10";
- github.com spring-vaadin repository, Issue: "Refreshing a view with a session scoped component throws error No child node found with id -1 #5229";
- github.com spring-vaadin repository, Issue: " Automatically check for incompatible scopes for Vaadin components #288"

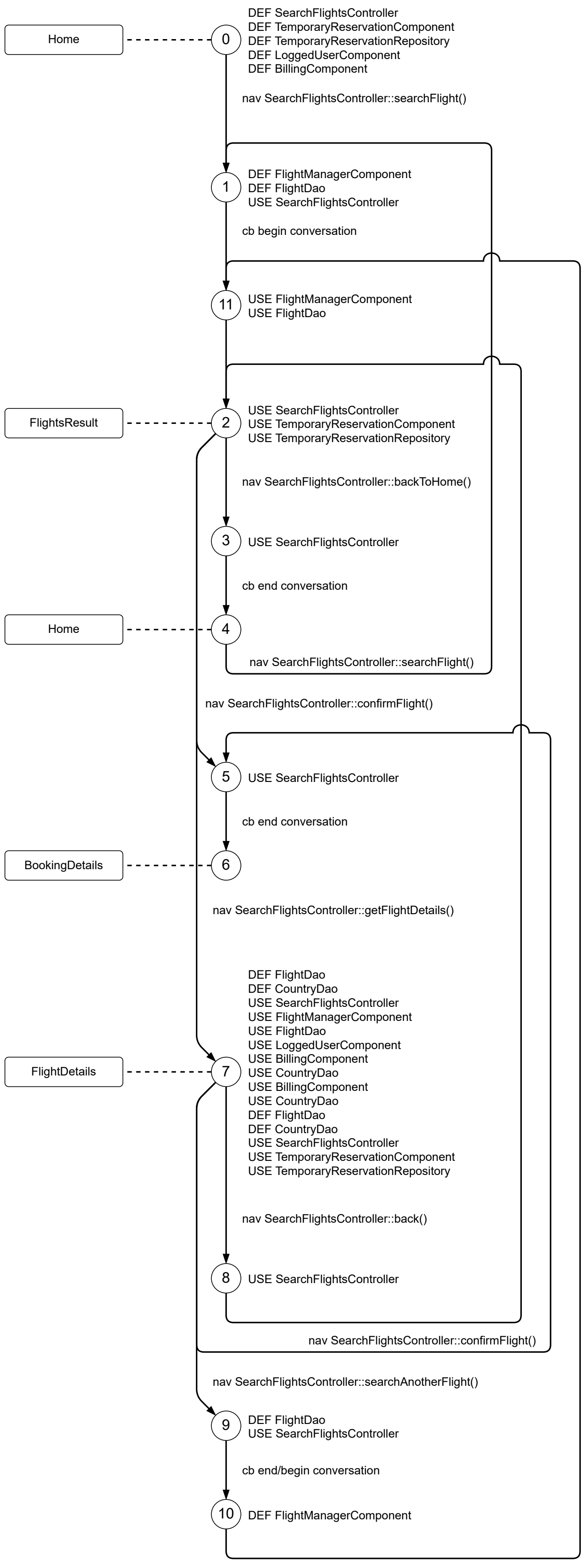
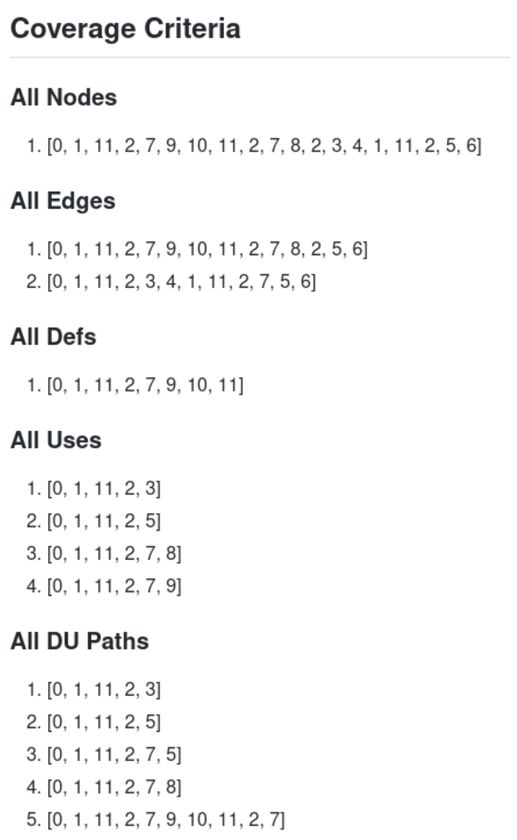

Home
0
DEF SearchFlightsController
DEF TemporaryReservationComponent
DEF TemporaryReservationRepository
DEF LoggedUserComponent
DEF BillingComponent
nav SearchFlightsController::searchFlight()
1
DEF FlightManagerComponent
DEF FlightDao
USE SearchFlightsController
cb begin conversation
11
USE FlightManagerComponent
USE FlightDao
FlightsResult
2
USE SearchFlightsController
USE TemporaryReservationComponent
USE TemporaryReservationRepository
nav SearchFlightsController::backToHome()
3
USE SearchFlightsController
cb end conversation
Home
4
nav SearchFlightsController::searchFlight()
nav SearchFlightsController::confirmFlight()
5
USE SearchFlightsController
cb end conversation
BookingDetails
6
nav SearchFlightsController::getFlightDetails()
DEF FlightDao
DEF CountryDao
USE SearchFlightsController
USE FlightManagerComponent
USE FlightDao
USE LoggedUserComponent
USE BillingComponent
USE CountryDao
USE BillingComponent
USE CountryDao
DEF FlightDao
DEF CountryDao
USE SearchFlightsController
USE TemporaryReservationComponent
USE TemporaryReservationRepository
FlightDetails
7
nav SearchFlightsController::back()
8
USE SearchFlightsController
nav SearchFlightsController::confirmFlight()
nav SearchFlightsController::searchAnotherFlight()
9
DEF FlightDao
USE SearchFlightsController
cb end/begin conversation
10
DEF FlightManagerComponent

Figure A.2: managed component Data Flow Graph available on the Search Flight use case documentation page.



Coverage Criteria

All Nodes

1. [0, 1, 11, 2, 7, 9, 10, 11, 2, 7, 8, 2, 3, 4, 1, 11, 2, 5, 6]

All Edges

1. [0, 1, 11, 2, 7, 9, 10, 11, 2, 7, 8, 2, 5, 6]
2. [0, 1, 11, 2, 3, 4, 1, 11, 2, 7, 5, 6]

All Defs

1. [0, 1, 11, 2, 7, 9, 10, 11]

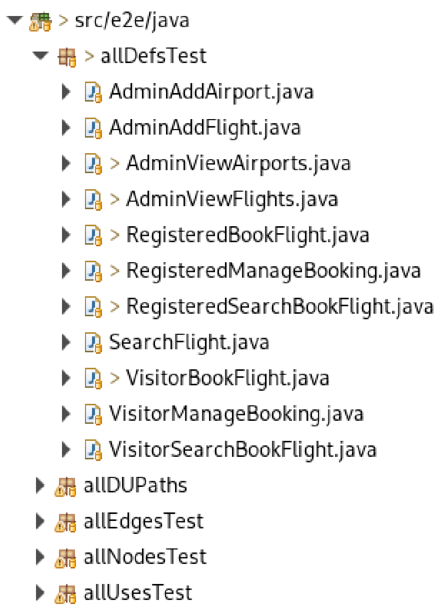
All Uses

1. [0, 1, 11, 2, 3]
2. [0, 1, 11, 2, 5]
3. [0, 1, 11, 2, 7, 8]
4. [0, 1, 11, 2, 7, 9]

All DU Paths

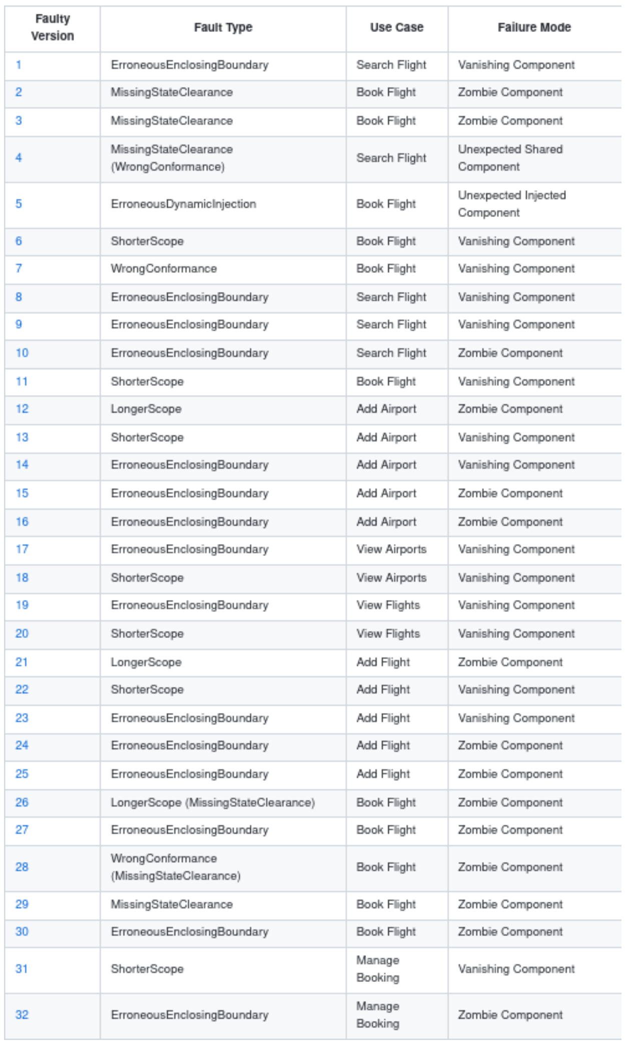
1. [0, 1, 11, 2, 3]
2. [0, 1, 11, 2, 5]
3. [0, 1, 11, 2, 7, 5]
4. [0, 1, 11, 2, 7, 8]
5. [0, 1, 11, 2, 7, 9, 10, 11, 2, 7]

Figure A.3: Coverage documentation at the Search Flight use case documentation page.



```
> src/e2e/java
  > allDefsTest
    AdminAddAirport.java
    AdminAddFlight.java
    > AdminViewAirports.java
    > AdminViewFlights.java
    > RegisteredBookFlight.java
    > RegisteredManageBooking.java
    > RegisteredSearchBookFlight.java
    SearchFlight.java
    > VisitorBookFlight.java
    VisitorManageBooking.java
    VisitorSearchBookFlight.java
  allDUPaths
  allEdgesTest
  allNodesTest
  allUsesTest
```

Figure A.4: Application test suites structure.



Faulty Version	Fault Type	Use Case	Failure Mode
1	ErroneousEnclosingBoundary	Search Flight	Vanishing Component
2	MissingStateClearance	Book Flight	Zombie Component
3	MissingStateClearance	Book Flight	Zombie Component
4	MissingStateClearance (WrongConformance)	Search Flight	Unexpected Shared Component
5	ErroneousDynamicInjection	Book Flight	Unexpected Injected Component
6	ShorterScope	Book Flight	Vanishing Component
7	WrongConformance	Book Flight	Vanishing Component
8	ErroneousEnclosingBoundary	Search Flight	Vanishing Component
9	ErroneousEnclosingBoundary	Search Flight	Vanishing Component
10	ErroneousEnclosingBoundary	Search Flight	Zombie Component
11	ShorterScope	Book Flight	Vanishing Component
12	LongerScope	Add Airport	Zombie Component
13	ShorterScope	Add Airport	Vanishing Component
14	ErroneousEnclosingBoundary	Add Airport	Vanishing Component
15	ErroneousEnclosingBoundary	Add Airport	Zombie Component
16	ErroneousEnclosingBoundary	Add Airport	Zombie Component
17	ErroneousEnclosingBoundary	View Airports	Vanishing Component
18	ShorterScope	View Airports	Vanishing Component
19	ErroneousEnclosingBoundary	View Flights	Vanishing Component
20	ShorterScope	View Flights	Vanishing Component
21	LongerScope	Add Flight	Zombie Component
22	ShorterScope	Add Flight	Vanishing Component
23	ErroneousEnclosingBoundary	Add Flight	Vanishing Component
24	ErroneousEnclosingBoundary	Add Flight	Zombie Component
25	ErroneousEnclosingBoundary	Add Flight	Zombie Component
26	LongerScope (MissingStateClearance)	Book Flight	Zombie Component
27	ErroneousEnclosingBoundary	Book Flight	Zombie Component
28	WrongConformance (MissingStateClearance)	Book Flight	Zombie Component
29	MissingStateClearance	Book Flight	Zombie Component
30	ErroneousEnclosingBoundary	Book Flight	Zombie Component
31	ShorterScope	Manage Booking	Vanishing Component
32	ErroneousEnclosingBoundary	Manage Booking	Zombie Component

Figure A.5: Table describing faulty versions of *Flight Manager*.

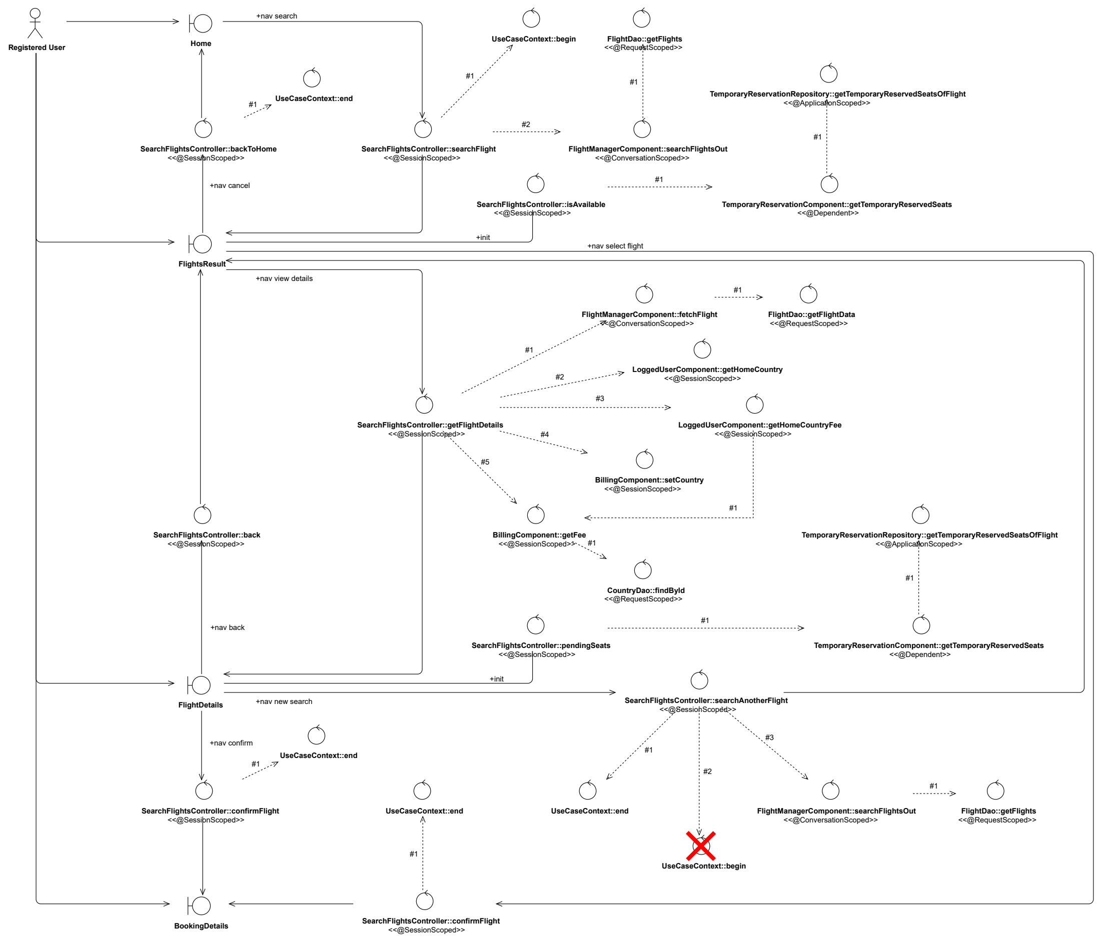
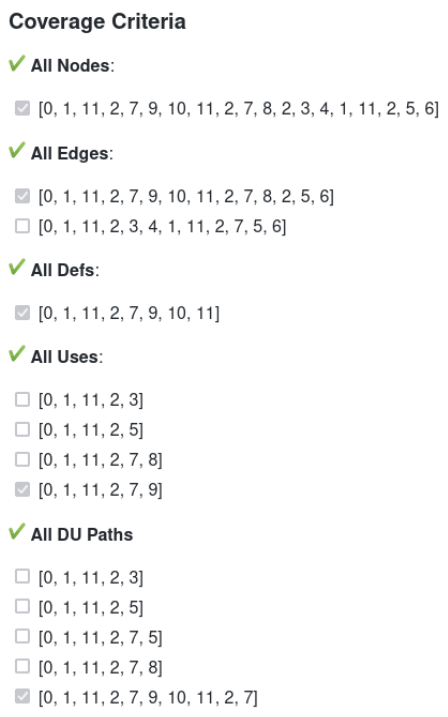


Figure A.6: Table describing faulty versions of *Flight Manager*.



Coverage Criteria

✓ **All Nodes**:

- [✓] [0, 1, 11, 2, 7, 9, 10, 11, 2, 7, 8, 2, 3, 4, 1, 11, 2, 5, 6]

✓ **All Edges**:

- [✓] [0, 1, 11, 2, 7, 9, 10, 11, 2, 7, 8, 2, 5, 6]
- [ ] [0, 1, 11, 2, 3, 4, 1, 11, 2, 7, 5, 6]

✓ **All Defs**:

- [✓] [0, 1, 11, 2, 7, 9, 10, 11]

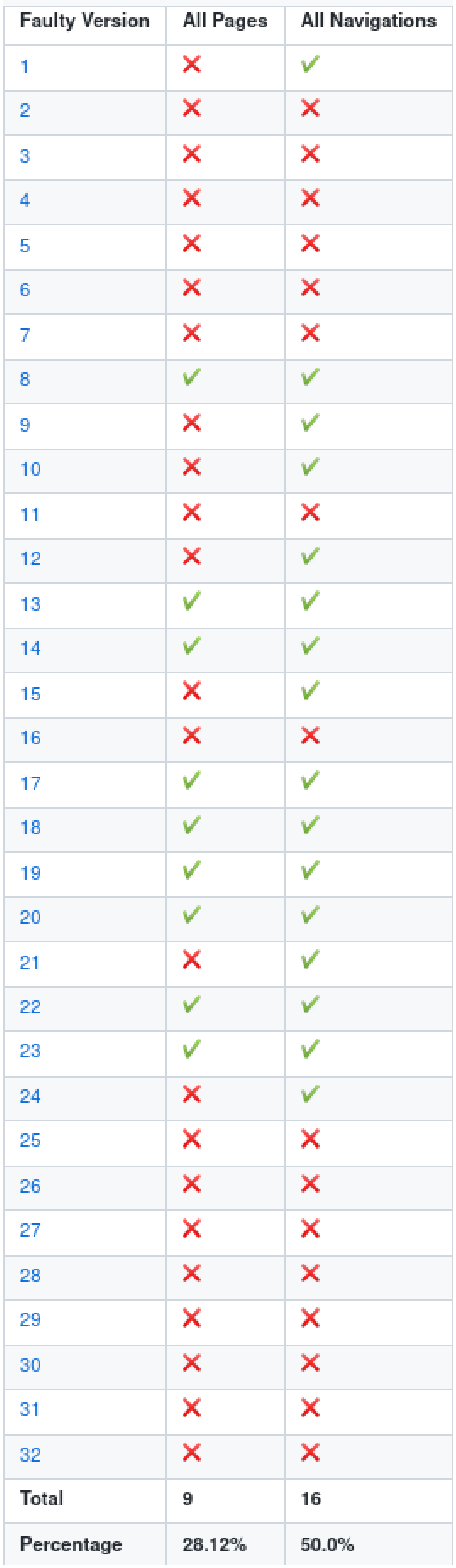
✓ **All Uses**:

- [ ] [0, 1, 11, 2, 3]
- [ ] [0, 1, 11, 2, 5]
- [ ] [0, 1, 11, 2, 7, 8]
- [✓] [0, 1, 11, 2, 7, 9]

✓ **All DU Paths**

- [ ] [0, 1, 11, 2, 3]
- [ ] [0, 1, 11, 2, 5]
- [ ] [0, 1, 11, 2, 7, 5]
- [ ] [0, 1, 11, 2, 7, 8]
- [✓] [0, 1, 11, 2, 7, 9, 10, 11, 2, 7]

Figure A.7: Test report of *Faulty Version* #1.



Faulty Version	All Pages	All Navigations
1	✗	✓
2	✗	✗
3	✗	✗
4	✗	✗
5	✗	✗
6	✗	✗
7	✗	✗
8	✓	✓
9	✗	✓
10	✗	✓
11	✗	✗
12	✗	✓
13	✓	✓
14	✓	✓
15	✗	✓
16	✗	✗
17	✓	✓
18	✓	✓
19	✓	✓
20	✓	✓
21	✗	✓
22	✓	✓
23	✓	✓
24	✗	✓
25	✗	✗
26	✗	✗
27	✗	✗
28	✗	✗
29	✗	✗
30	✗	✗
31	✗	✗
32	✗	✗
Total	**9**	**16**
Percentage	**28.12%**	**50.0%**

Figure A.8: Fault detection capabilities of *All Pages* and *All Navigation* coverage criteria.

Appendix B

Publications

Peer reviewed conference papers

1. L. Carnevali, M. Paolieri, R. Reali, **L. Scommegna**, F. Tammaro, E. Vicario, " Using the ORIS tool and the SIRIO library for model driven engineering of quantitative analytics ", *European Performance Engineering Workshop* , 2022

Workshop papers

1. J. Parri, **L. Scommegna**, S. Sampietro, E. Vicario, " Evaluation of software aging in component-based Web Applications subject to soft errors over time ", *WoSAR: International Workshop on Software Aging and Rejuvenation* , 2021

2. L. Carnevali, M. Paolieri, R. Reali, **L. Scommegna**, E. Vicario, " A Markov Regenerative Model of Software Rejuvenation Beyond the Enabling Restriction ", *WoSAR: International Workshop on Software Aging and Rejuvenation* , 2022

Papers under review

1. **L. Scommegna**, J. Parri, S. Sampietro, E. Vicario, " Model-Based Testing of dependency injection and automated lifecycle management in stateful Web Applications", *IEEE Transactions on Software Engineering*

2. N. Bertocci, L. Carnevali, **L. Scommegna**, E. Vicario, " Efficient derivation of optimal semaphore schedules for multimodal urban intersections.", *IEEE Transactions on Intelligent Transportation Systems*

Bibliography

Allen, F. E. (1970). Control flow analysis. *ACM Sigplan Notices*.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Alonso, J., Matias, R., Vicente, E., Maria, A., and Trivedi, K. S. (2013). A comparative experimental study of software rejuvenation overhead. *Performance Evaluation*, 70(3):231–250.

Ament, J. D. (2013). *Arquillian Testing Guide*. Packt Publishing.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., Mcminn, P., Bertolino, A., et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001.

Andrade, E. and Machida, F. (2019). Analysis of software aging impacts on plant anomaly detection with edge computing. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 204–210. IEEE.

Avižienis, A., Laprie, J.-C., and Randell, B. (2004). Dependability and its threats: a taxonomy. In *Building the Information Society*, pages 91–120. Springer.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

Avritzer, A., Pietrantuono, R., and Trivedi, K. (2020). Future directions for software aging and rejuvenation research. In *Handbook Of Software Aging And Rejuvenation: Fundamentals, Methods, Applications, And Future Directions*, pages 355–362.

Azizimazreah, A., Gu, Y., Gu, X., and Chen, L. (2018). Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE.

Barth, A. (2011). Rfc 6265-http state management mechanism. *Internet Engineering Task Force (IETF)*, pages 2070–1721.

Bartocci, E., Falcone, Y., Francalanza, A., and Reger, G. (2018). Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer.

Baumann, R. C. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316.

Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.

Bondavalli, A. and Simoncini, L. (1990). Failure classification with respect to detection. In *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–53. IEEE.

Booch, G. (2005). *The unified modeling language user guide*. Pearson Education India.

Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2007). A subset of precise uml for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104.

Brown, K., Craig, G., Hester, G., Amsden, J., Berg, D., Pitt, D., Stinehour, R., Jakab, P. M., and Weitzel, M. (2003). *Enterprise Java Programming with IBM WebSphere*. Addison-Wesley Professional.

Buschmann, F., Henney, K., and Schmidt, D. C. (2007). *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*, volume 4. John Wiley & Sons New York, NY, USA.

Cerveira, F., Oliveira, R. A., Barbosa, R., and Madeira, H. (2020). Evaluation of restful frameworks under soft errors. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 369–379. IEEE.

Cockburn, A. (2001). *Writing effective use cases*. Pearson Education India.

Conallen, J. (1999). Modeling web application architectures with uml. *Communications of the ACM*, 42(10):63–70.

Conallen, J. (2003). *Building Web applications with UML*. Addison-Wesley Professional.

Cotroneo, D., Fucci, F., Iannillo, A. K., Natella, R., and Pietrantuono, R. (2016). Software aging analysis of the android mobile os. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*, pages 478–489. IEEE.

Cotroneo, D., Natella, R., and Pietrantuono, R. (2013). Predicting aging-related bugs using software complexity metrics. *Performance Evaluation*, 70(3):163–178.

Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2011). Software aging and rejuvenation: Where we are and where we are going. In *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*, pages 1–6. IEEE.

Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2014). A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(1):1–34.

Crocker, D. et al. (1982). Standard for the format of arpa internet text messages.

Curcio, K., Navarro, T., Malucelli, A., and Reinehr, S. (2018). Requirements engineering: A systematic mapping study in agile software development. *Journal of Systems and Software*, 139:32–50.

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

Denaro, G., Gorla, A., and Pezzè, M. (2008). Contextual integration testing of classes. In *International Conference on Fundamental Approaches to Software Engineering*, pages 246–260. Springer.

Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.

Dohi, T., Trivedi, K. S., and Avritzer, A. (2020). *Handbook of Software Aging and Rejuvenation: Fundamentals, Methods, Applications, and Future Directions*. World Scientific.

dos Santos, F. F., Draghetti, L., Weigel, L., Carro, L., Navaux, P., and Rech, P. (2017). Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE.

Escobar, L. A. and Meeker, W. Q. (2006). A review of accelerated test models. *Statistical science*, pages 552–577.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

Fioravanti, S., Mattolini, S., Patara, F., and Vicario, E. (2016). Experimental performance evaluation of different data models for a reflection software architecture over nosql persistence layers. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 297–308.

Foote, B., Rohnert, H., and Harrison, N. (1999). *Pattern Languages of Program Design 4*. Addison-Wesley Longman Publishing Co., Inc., USA.

Fowler, M. (2012). *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley.

Frankl, P. G. and Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498.

Frost, B. (2016). *Atomic design*. Brad Frost Pittsburgh.

Fuentes-Fernández, L. and Vallecillo-Moreno, A. (2004). An introduction to uml profiles. *UML and Model Engineering*, 2(6-13).

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical Report CMU/SEI-94-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Godbolt, M. (2016). *Frontend architecture for design systems: a modern blueprint for scalable and sustainable websites*. " O'Reilly Media, Inc.".

Gojare, S., Joshi, R., and Gaigaware, D. (2015). Analysis and design of selenium webdriver automation testing framework. *Procedia Computer Science*, 50:341–346.

Grottke, M., Matias, R., and Trivedi, K. S. (2008). The fundamentals of software aging. In *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)*, pages 1–6. Ieee.

Grottke, M. and Trivedi, K. S. (2007). Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2).

Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. *ACM SIGSOFT Software Engineering Notes*, 19(5):154–163.

Hat, J. R. (2016). Shrinkwrap: Java api for archive manipulation.

Kaplan, M., Klinger, T., Paradkar, A. M., Sinha, A., Williams, C., and Yilmaz, C. (2008). 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.

Kishani, M., Tahoori, M., and Asadi, H. (2019). Dependability analysis of data storage systems in presence of soft errors. *IEEE Transactions on Reliability*, 68(1):201–215.

Koutras, V. P. and Platis, A. N. (2006). Applying software rejuvenation in a two node cluster system for high availability. In *2006 International Conference on Dependability of Computer Systems*, pages 175–182. IEEE.

Kuliamin, V. V., Petrenko, A. K., Kossatchev, A. S., and Burdonov, I. B. (2003). The unitesk approach to designing test suites. *Programming and Computer Software*, 29(6):310–322.

Kung, D. C., Liu, C.-H., and Hsia, P. (2000). An object-oriented web test model for testing web applications. In *Proceedings First Asia-Pacific Conference on Quality Software*, pages 111–120. IEEE.

Legeard, B. and Utting, M. (2010). Model-based testing-next generation functional software testing. *SoftwareTech News*, 12(4).

Limon, S., Yadav, O. P., and Liao, H. (2017). A literature review on planning and analysis of accelerated testing for reliability assessment. *Quality and Reliability Engineering International*, 33(8):2361–2383.

Liu, C.-H., Kung, D. C., Hsia, P., and Hsu, C.-T. (2000). Structural testing of web applications. In *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. IEEE.

Machida, F., Kim, D. S., and Trivedi, K. S. (2010). Modeling and analysis of software rejuvenation in a server virtualized system. In *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*, pages 1–6. IEEE.

Madeira, H., Costa, D., and Vieira, M. (2000). On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 417–426. IEEE.

Manuel, P. D. and AlGhamdi, J. (2003). A data-centric design for n-tier architecture. *Information Sciences*, 150(3-4).

Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34):597.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA.

May, T. C. and Woods, M. H. (1978). A new physical mechanism for soft errors in dynamic memories. In *16th International Reliability Physics Symposium*, pages 33–40. IEEE.

May, T. C. and Woods, M. H. (1979). Alpha-particle-induced soft errors in dynamic memories. *IEEE transactions on Electron devices*, 26(1):2–9.

McDermid, J. and Pumfrey, D. (1994). A development of hazard analysis to aid software design. In *Proceedings of COMPASS'94-1994 IEEE 9th Annual Conf. on Computer Assurance*, pages 17–25. IEEE.

Meeker, W. Q. and Escobar, L. A. (1993). A review of recent research and current issues in accelerated testing. *International Statistical Review/Revue Internationale de Statistique*, pages 147–168.

Menasché, D., Trivedi, K., and Altman, E. (2019). Rejuvenation and the age of information. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 225–231. IEEE.

Moore, K. and Freed, N. (2000). Use of http state management. Technical report, RFC 2964, Network Working Group.

Natella, R., Cotroneo, D., Duraes, J. A., and Madeira, H. S. (2012). On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96.

Nebut, C., Fleurey, F., Le Traon, Y., and Jezequel, J.-M. (2006). Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155.

Offutt, J. and Abdurazik, A. (1999). Generating tests from uml specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer.

Patara, F. and Vicario, E. (2014). An adaptable patient-centric electronic health record system for personalized home care. In *2014 8th International Symposium on Medical Information and Communication Technology (ISMICT)*, pages 1–5. IEEE.

Pérez-Castillo, R., De Guzman, I. G.-R., and Piattini, M. (2011). Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4):367–375.

Ricca, F. and Tonella, P. (2001). Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34. IEEE.

Richardson, C. (2006). *POJOs in Action, Developing Enterprise Applications with Lightweight Frameworks*. Manning.

Rosenberg, D. and Scott, K. (1999). *use case driven object modeling with uml*. Springer.

Rosenberg, D., Stephens, M., and Collins-Cope, M. (2005). Agile development with iconix process. *New York, Editorial Apress*.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society*, 39(2):25.

Sharma, T. and Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, 138:158–173.

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

Souter, A. L. and Pollock, L. L. (2003). The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018.

Souter, A. L., Pollock, L. L., and Hisley, D. (1999). 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*, pages 47–56.

Souza, V. E. S., Falbo, R., and Guizzardi, G. (2007). A uml profile for modeling framework-based web information systems. In *12th International Workshop on Exploring Modelling Methods in Systems Analysis and Design EMMSAD*, volume 782007, pages 153–162.

Sundaram, V., Creti, M. T., Panta, R. K., and Bagchi, S. (2008). Component-dependency based micro-rejuvenation scheduling. In *Fast Abstract in the Supplemental Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Anchorage, Alaska, USA*. Citeseer.

Tiwari, S. and Gupta, A. (2015). A systematic literature review of use case specifications research. *Information and Software Technology*, 67:128–158.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312.

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., and Kazmeier, J. (2006). Automation of gui testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14.

Wu, Y., Pan, D., and Chen, M.-H. (2000). Techniques of maintaining evolving component-based software. In *Proceedings 2000 International Conference on Software Maintenance*, pages 236–246. IEEE.

Xiang, J., Weng, C., Zhao, D., Andrzejak, A., Xiong, S., Li, L., and Tian, J. (2019). Software aging and rejuvenation in android: new models and metrics. *Software Quality Journal*, pages 1–22.

Yacoub, S., Cukic, B., and Ammar, H. H. (2004). A scenario-based reliability analysis approach for component-based software. *IEEE transactions on reliability*, 53(4):465–480.