Stateful Software Systems: Unraveling the Complexities of Transient Data Management

Laura Carnevali **Benedetta Picano** **Riccardo Reali** **Leonardo Scommegna**
Roberto Verdecchia **Enrico Vicario**

Dept. of Information Engineering, University of Florence

Software Technologies Lab - https://stlab.dinfo.unifi.it

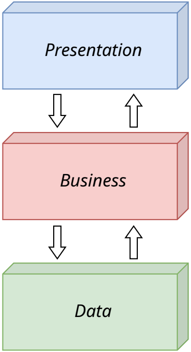
QualITA'24

Venice, June 2024

- this is about:
  - Transient data management in software systems
  - How transient data identifies an underlying concurrent process
  - What are the consequences in terms of reliability
  - Some strategies to improve reliability

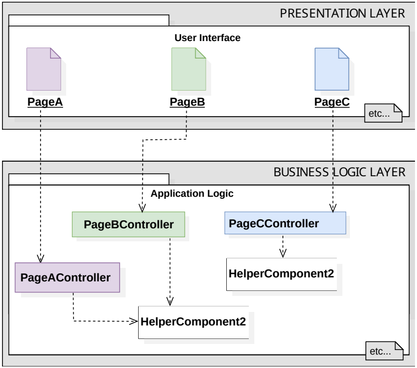
Software System Common Oganization

- A client interacts with the system through the user interface;
- The presentation layer converts the interaction in an *input* for the Business Layer;
- The business layer, starts an elaborating process possibly encompassing the data layer;
- Once finished, the business layer forwards the response to the presentation layer;



Business Logic Composition

- **Controllers**:
  - Implement *Page* or *View Controller* pattern [1]
  - Responsible for inputs from a specific page or from the entire application
- **Helper components**:
  - Provide auxiliary services
  - Usually injected in dependent components (dependency injection pattern)
  - Can be shared among multiple components



[1] *Buschmann, Henney and Schmidt, "Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing", Volume 4. 2007.*

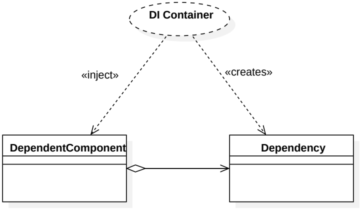
Stateful Business Transactions

- Use cases can not be always **stateless** business transactions
- **Session state**[2]: a state with a transient nature usually stored in-memory
- Example: the shopping cart in an e-commerce web application
- Business logic components take care of session state management
- A stateful business transaction **implies** a stateful application business logic
- Business logic **components become stateful**
- Although necessary, stateful business logic requires a higher level of **complexity**

[2] *Fowler, Martin, "Patterns of Enterprise Application Architecture" Addison-Wesley 2012*

Dependency Injection Frameworks

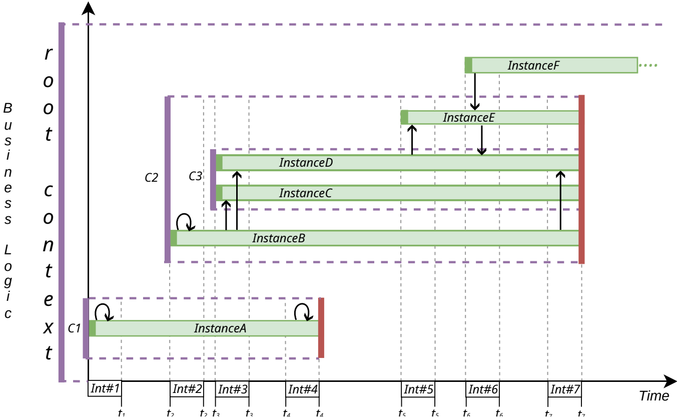
- **DI container** responsibilities:
  - **Creates** the dependency component
  - **Injects** the dependency in the client component
  - **Destroys** the dependency when no longer needed
  - Implements an **automatic life cycle management** mechanism
- Rely on **Visibility Context** concept
- Pervasive paradigm considered a **best practice**
- Main challenges addressed:
  - **Scalability**: automatic resolution through meta-information
  - **Stateful dependency injection**: achieved through visibility contexts



Examples of DI frameworks

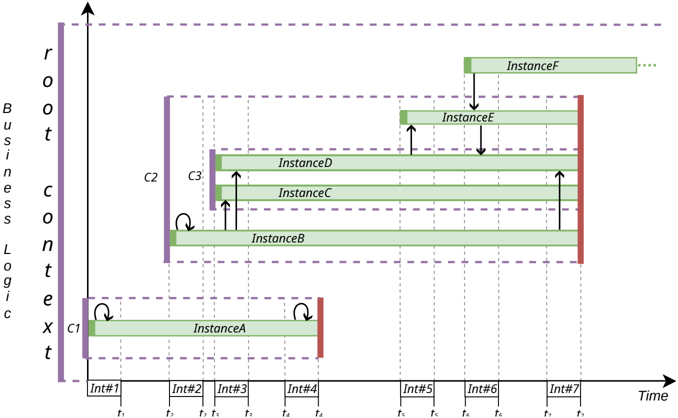
- **Context and Dependency Injection (CDI)**:
  - Part of the Java/Jakarta Enterprise Edition (JEE) set of specifications
  - Popular framework to manage backend-side application logic
    i.e., Stateful Architectures
  - Contexts shaped by the HTTP: application, request, and session scope
- **Angular**:
  - Popular framework to manage client-side application logic
    i.e., Service Oriented Architectures
  - User interactions on the interface mark the context life cycle
  - Life cycle usually tied to the life cycle of a UI widget
  - Note that a widget can be composed of multiple widgets (composite structure)

Business Logic Runtime Evolution



- **X-axis:** requests arriving over (continuous) time
- **Y-axis:** Instances (green) and Contexts (purple)
- Set of alive stateful instances constitutes the **internal state** of the system

Business Logic Runtime Evolution



- Internal state evolves over time as a result of:
  - **Application logic** defined at static time by the code
  - **Sequence of interactions** issued by the user at runtime
  - Rules and mechanisms of the third party **DI framework**

Downsides of Transient Data Management

- Behavior of the system depends on its **internal state**
- **Challenge**: predict the evolution of the internal state and its effects is hard
- Aggregation of components with different lifecycles reduces **designer control**
- Considering the effect of all the possible input sequences is **unfeasible**
- DI frameworks include **additional opacity** to the state evolution process
- **Testability**: business logic usually tested without the DI container

Fault Model

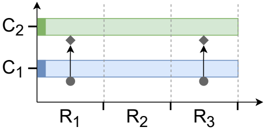
- Taxonomy of fault types:
  - **ShorterScope**
  - **LongerScope**
  - **WrongConformance**
  - **EarlyOrUndueClosure**
  - **LateOrMissingClosure**
  - **LateOrMissingBegin**
  - **MissingStateClearance**
  - **ErroneousDynamicInjection**
- Reflect structural characteristics of managed components
- Covers major complexities and issues
  - Observed during the STLab Development Experience
  - Reported by developers of different levels of skills
  - Posted on technical social forums (e.g., Stack Overflow, GitHub)

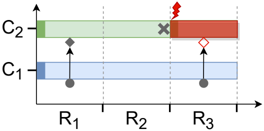
Failure Modes

- Faults may result in various kinds of errors in the sw components
- Errors may eventually cause various types of deviations in the functional behavior delivered by the UI:
    - **Vanishing Component**: An injected component may not live and maintain its state with continuity along the time interval needed by its dependants
    - **Zombie Component**: an injected component may remain alive with continuity while a dependent component expects that it is destroyed and restarted
    - **Unexpected Shared Component**: A context may remain continuously active so as to be accessible by two or more concurrent dependent contexts
    - **Unexpected Injected Component**: The type of a required component may be wrongly specified at its injection point

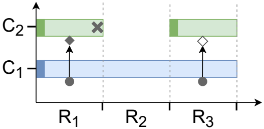
Vanishing and Zombie Component: *correct vs faulty behavior*

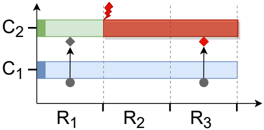
- Vanishing Component:





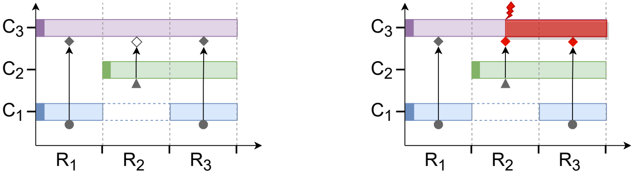
- Zombie Component:



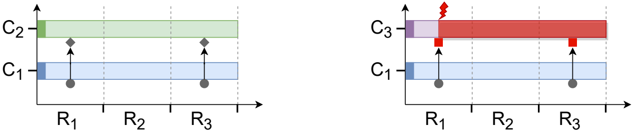


Unexpected Shared and Injected Component: *correct vs faulty behavior*

- Unexpected Shared Component:

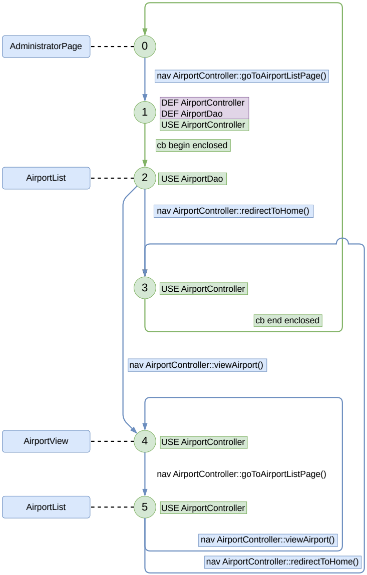


- Unexpected Injected Component:



Fighting Faults through Model-Based Testing[3]

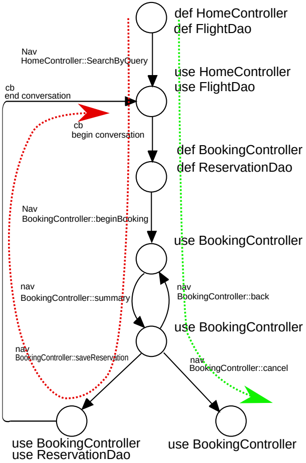
- **Managed Component Data Flow Graph** Abstraction (mcDFG)
- Aware of:
    - Application logic
    - Navigation design
    - DI container mechanisms
- Aimed to find faults causing identified failures
- Test case selection guided by various **coverage criteria**


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()

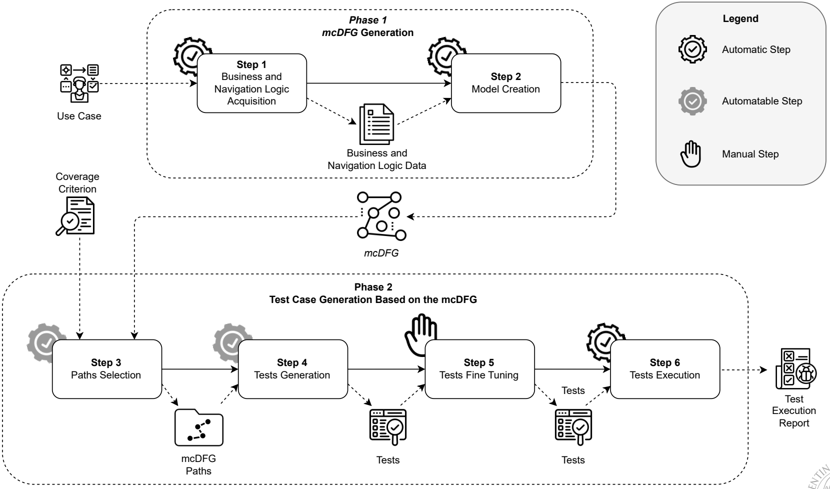
[3] *Scommegna, Verdecchia, Vicario. Unveiling Faulty User Sequences: A Model-Based Approach to Test Three-Tier Software Architectures. JSS, 2024*

Details of Paths Selection

- Each **mcDFG Path**:
    - Suggests a Test Case
    - Represents a sequence of user interaction
    - Triggers a specific business logic and DI container behavior
- A **Coverage Criteria** suggests a Test Suite
- **Gray Box** Testing:
    - Web Driver simulate User Interaction
    - Assertion may concern UI or Application State
- Implemented with **Arquillian Warp** and **Selenium WebDriver**



Complete Testing Methodology[4]



[4] *Scommegna, Verdecchia, Vicario. Unveiling Faulty User Sequences: A Model-Based Approach to Test Three-Tier Software Architectures. JSS, 2024*

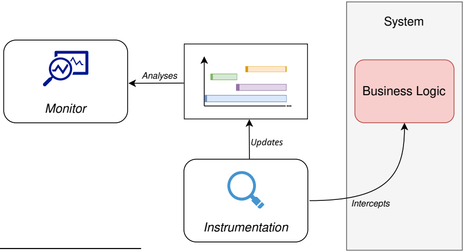
Results

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

- **Feasible Number** of Tests even for expensive coverage criteria: a few tens of test cases in the worst cases
- **Good Fault Detection Capacity**: so far DFG has always detected the injected fault
- Test case generation is fast once the initial setup is configured

Discussion and Conclusion

- Evolution of the system state as a concurrent process
- Fault hidden in code captured through ad-hoc **MBT methodology**[5]
- Lifecycle management as partial **Software Rejuvenation** of the system state[6]
- Runtime **extraction of the concurrent process** with a minimal intrusive instrumentation tool [7]
- **Ongoing direction**: using the extractor to implement a **Runtime Verification** framework



[5] *Scommegna, Verdecchia, Vicario. Unveiling Faulty User Sequences: A Model-Based Approach to Test Three-Tier Software Architectures. JSS, 2024*

[6] *Parri, Sampietro, Scommegna, Vicario. Evaluation of software aging in component-based web applications subject to soft errors over time, WoSAR, 2021*

[7] *Scommegna, Picano, Verdecchia, Vicario. OREO: A Tool-Supported Approach for Offline Run-time Monitoring and Fault-Error-Failure Chain Localization, STVR Under Revision*