**People's Democratic Republic of Algeria**
**Ministry of Higher Education and Scientific Research**

---

**Abou Bekr Belkaid University - Tlemcen**


**Faculty of Sciences**
**Department of Computer Science**

**Final study thesis for obtaining a Master's degree in Computer Science**

**Option: Software Engineering**

# Subject

# Systemic Approach to Categorizing and Modeling Requirements

## Development of a Support Tool for the SMART Method

**Prepared by:**
**BOUKARABILA Rihab Wahiba**

**Presented on June 29, 2025 before the jury composed of:**

| | |
|---|---|
| **Mrs. Fassila Lamya BENMANSOUR** | President (University of Tlemcen) |
| **Mr. Amine BRIKCI-NIGASSA** | Examiner (University of Tlemcen) |
| **Prof. Azeddine CHIKH** | Supervisor (University of Tlemcen) |

**Academic Year: 2024 - 2025**

*"Whoever thinks that they are independent of knowledge is ignorant, and whoever claims to have knowledge has indeed become ignorant."*
*—Ibn al-Muqaffa*

# Dedication

To my parents, thank you for your unconditional love, your prayers, and your endless sacrifices. You are my greatest inspiration.

To my siblings: Siham, Youcef, and Sarah, thank you for your love, encouragement, and unwavering support at every step of this journey.

To my friends, thank you for the shared moments, the laughter, and the memories that made everything feel lighter.

To my university professors, thank you for your guidance, your dedication to teaching, and for creating an environment where I could grow and thrive.

And to everyone who supported me through words, presence, or silent encouragement, I am deeply grateful.

# Acknowledgments

First and foremost, I am extremely grateful to Allah for giving me health and courage and determination throughout all my study years.

I extend my deepest gratitude to my supervisor, Professor Azeddine CHIKH, for his patience, insightful guidance, and support. Working under his supervision has been both an honor and a learning experience I will always value.

I am also sincerely appreciative of Mrs. BENMANSOUR Fassila Lamya for presiding over my thesis jury, and I deeply value the time and attention she devoted to this process.

My sincere appreciation also goes to Mr. BRIKCI-NIGASSA Amine for agreeing to examine my work and provide his expertise during this defense.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **BS2** | Business System Software |
| **CI/CD** | Continuous Integration / Continuous Deployment |
| **E2** | Endogenous Exogenous |
| **FR** | Functional Requirement(s) |
| **GF** | Generic Factor |
| **GS** | Generic Subject |
| **MVC** | Model-View-Controller |
| **NFR** | Non-Functional Requirement(s) |
| **RE** | Requirements Engineering |
| **RML** | Requirements Metadata Language |
| **R2F** | Referential Requirement Framework |
| **SMART** | Systemic Approach of caTegorizing and Modeling Requirements |
| **SF** | Specific Factor |
| **SRS** | Software Requirements Specification |
| **SS** | Specific Subject |
| **UA** | User Analyst |
| **UML** | Unified Modeling Language |

# I / Introduction

In an era where software systems permeate every aspect of human activity—from business and governance to education and healthcare—the complexity of delivering software that truly meets user needs has grown exponentially. These systems are no longer developed in isolated contexts; they are shaped by dynamic environments, evolving technologies, and diverse stakeholder expectations. As software becomes more critical, the ability to define, structure, and manage requirements effectively is no longer a luxury—it is a foundational necessity for success.

Requirements Engineering (RE), as a discipline, seeks to formalize this essential task. Yet, despite its evolution and the existence of various frameworks and methodologies, the reality in most real-world projects remains far from ideal. Practitioners continue to face deeply rooted challenges: vague and ambiguous requirement statements, inconsistent categorization, difficulties in traceability, and the struggle to translate informal stakeholder inputs into clear, structured, and actionable specifications. These gaps hinder the very goals RE is meant to serve: clarity, consistency, and alignment with actual user needs.

Traditional classifications, such as the common functional vs. non-functional dichotomy, often fall short when faced with the multi-layered, domain-sensitive nature of modern software. These methods rarely capture the full semantic richness required for advanced modeling and reuse. Moreover, the disconnect between what users express and what analysts formalize continues to pose significant risks to software quality, user satisfaction, and project timelines.

In response to these persistent issues, Professor Azeddine Chikh proposed the "Systemic Approach of caTegorizing and Modeling Requirements" [6].

This method introduces a structured and multidimensional approach to requirement modeling, built around two central pillars: the R2F Framework, which defines a comprehensive space for organizing requirement subjects and influencing factors; and a hybrid categorization model that combines multiple classification axes, including the source of the requirement, its abstraction level, and its relation to internal or external system conditions.

While SMART provides a robust theoretical foundation, its practical adoption has been limited by the lack of supporting digital tools. Without a dedicated platform to assist experts and analysts in navigating its rich conceptual space, the method remains largely academic. Addressing this need is the core motivation of this thesis.

This work presents the design and implementation of the Expert Dashboard a web-based module that operationalizes the SMART method from the expert's perspective. It enables structured management of requirement subjects, influencing factors, and system constraints through a guided interface aligned with the R2F model. The Expert Dashboard serves as the first step toward a larger suite of tools intended to bridge the gap between theoretical rigor and practical applicability in requirements engineering.

By grounding abstract principles in an interactive, usable system, this thesis not only demonstrates the feasibility of SMART in real-world settings but also lays the foundation for future extensions. These may include interfaces for analysts, collaborative features, and advanced traceability mechanisms, ultimately contributing to a more robust and intelligent RE ecosystem.

# II / Requirements Engineering

## CONTENT

# II.1 Introduction

In a landscape marked by agile development, continuous delivery, and accelerating technological change, clear and well-defined requirements have become a cornerstone of successful software systems.
Poorly defined or misunderstood requirements often lead to budget overruns, delayed timelines, and systems that ultimately fail to meet user expectations.In contrast, well-articulated requirements provide a solid foundation for design, implementation, and validation. However, achieving such clarity is inherently challenging due to the diverse perspectives of stakeholders, evolving business needs, and the abstract nature of software itself. To address these challenges, the discipline of Requirements Engineering (RE) has emerged as a structured approach to identifying, documenting, and managing software requirements throughout the development lifecycle.

RE serves two primary objectives: first, to develop a deep understanding of the problem that the intended software system aims to address; and second, to identify, select, and clearly document the system's functional and non-functional requirements, along with constraints related to its development. These outcomes form the foundation for all subsequent phases of the software development lifecycle, guiding design decisions, implementation efforts, testing strategies, and stakeholder communication. A well-executed RE process not only improves alignment between system behavior and stakeholder expectations but also reduces the risk of costly rework and project failure.

As software systems continue to evolve, the discipline of Requirements Engineering faces new challenges, including:

- **Ambiguous or Incomplete Requirements:** Stakeholders often use vague terms or omit details, leading to misinterpretations and unmet expectations.
- **Conflicting Priorities:** Stakeholders may have opposing goals, making it difficult to define a unified and consistent set of requirements.
- **Communication Gaps:** Misunderstandings between stakeholders and developers can result in features that fail to reflect actual needs.
- **Alignment with Agile Methods:** Agile's preference for minimal documentation can clash with RE's emphasis on detailed specification, creating tension between speed and completeness.
- **Evolving Requirements:** In dynamic environments, requirements often change mid-project. Managing these changes while preserving consistency and traceability remains a major RE challenge.

Addressing these challenges requires more than just technical solutions—it demands a shared conceptual foundation. A clear understanding of the core principles, terminology, and classifications within Requirements Engineering (RE) is essential for ensuring consistency, traceability, and stakeholder alignment throughout the development process.

## II.2 Definitions

Before delving into the methodologies and classifications explored in this thesis, it is essential to clarify the foundational concepts of Requirements Engineering (RE).

### II.2.1 Requirements Engineering

Requirements Engineering (RE) is defined by the IEEE/ISO/IEC 29148:2018 standard as: [14]

> "An interdisciplinary function that mediates between the domains of the acquirer and supplier or developer to establish and maintain the requirements to be met by the system, software, or service of interest."

### II.2.2 Requirements

Additionally, Sommerville [26] describes requirements as:

> "A specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute."

### II.2.3 Functional Requirements

**Functional requirements** define what the system should do—such as processing data, performing calculations, or handling user input. They specify the core behavior of the system and are typically represented through use cases, workflows, or feature descriptions.

They are generally testable and verifiable, making them essential during the design, implementation, and validation phases of development [26, 14].

### II.2.4 Non-Functional Requirements

**Non-functional requirements** (NFRs) describe how the system should perform rather than what it should do. These include attributes like performance, security, reliability, usability, and maintainability [7, 19, 22, 26].

### II.2.5 Stakeholders

A **stakeholder** is any person, group, or organization affected by or capable of influencing the software system. This includes both internal actors (e.g., developers, project managers) and external ones (e.g., clients, end-users, regulators) [28].

Identifying and engaging stakeholders is critical to ensuring that system requirements reflect real-world needs and constraints.

### II.2.6 Software Requirements Specification (SRS)

A **Software Requirements Specification** is a formal document that defines the intended functionality, constraints, and context of a software system. It serves as a

bridge between stakeholders by translating informal needs into structured, testable, and traceable requirements [14].

According to the IEEE 29148:2018 standard, an SRS should capture both functional and non-functional requirements, ensuring alignment with stakeholder goals and system capabilities. A well-structured SRS typically includes system purpose, detailed requirements, use cases, and traceability links to design and testing artifacts [26].

### Volere

The **Volere** Requirements Specification Template, developed by Suzanne and James Robertson, is a structured framework for documenting and analyzing requirements. It guides the elicitation of both functional and non-functional needs through a well-defined template that emphasizes completeness, traceability, and stakeholder communication. Volere promotes clarity through features like *fit criteria* and reusable requirement patterns. It is methodology-agnostic and suitable for both agile and traditional processes [24].

### IEEE 830 SRS Template

The **IEEE 830** standard defines a recommended structure for writing Software Requirements Specifications (SRS). It provides a formal template that organizes requirements into sections such as introduction, overall system description, specific functional and non-functional requirements, and appendices. The goal is to ensure clarity, completeness, and traceability across stakeholders and development phases [12].

## II.2.7 Requirement Categorization

**Requirement categorization** is the process of classifying requirements based on attributes such as source, abstraction level, or operational context [19, 22].

## II.2.8 Requirement Modeling

**Requirement modeling** refers to representing requirements in structured or semi-formal forms—such as diagrams, templates, or conceptual frameworks—to support communication, analysis, and traceability [22, 15].

## II.2.9 Traceability

**Traceability** is the ability to track requirements across the software lifecycle—from initial specification to design, implementation, and testing [11, 9].

It supports change impact analysis and validation, and is typically divided into forward and backward traceability. The SMART framework enhances traceability by semantically organizing requirements, and the Expert Dashboard reinforces this through explicit context and classification links [22].

### II.2.10 Requirements Elicitation

**Requirements elicitation** involves gathering stakeholder needs, expectations, and constraints through techniques such as interviews, workshops, observation, or document analysis [33, 22].

### II.2.11 Software Development Life Cycle

The **Software Development Life Cycle (SDLC)** is a structured process that guides the development, deployment, and maintenance of software systems.

It ensures a systematic progression from initial concept to final release, aiming to deliver high-quality software that meets user and organizational needs.

SDLC typically includes phases such as *requirements analysis*, *planning*, *design*, *implementation*, *testing*, *deployment*, and *maintenance*. Each phase contributes to better control over scope, cost, quality, and risk.

Several SDLC models exist—Waterfall, V-model, Spiral, and Agile—each suited to different project contexts. While traditional models emphasize sequential development and documentation, modern approaches increasingly favor adaptability and iterative refinement.

#### Agile Development Methodology

**Agile** is a modern software development approach emphasizing iterative progress, stakeholder collaboration, and adaptability. It is organized into short cycles (*sprints*) that deliver functional software increments and enable continuous feedback [2].

Agile frameworks like Scrum and Kanban encourage early validation and responsiveness to change. While Agile reduces upfront documentation, it poses challenges for traceability and requirement stability.

## II.3 Existing Taxonomies in RE

Existing taxonomies in Requirements Engineering (RE) provide structured classification schemes that facilitate the organization, analysis, and understanding of requirements and related concepts. This section explores these classifications to highlight the dominant approaches used within the field.[19]

**Positioning** This thesis focuses on two key taxonomies central to structured requirement modeling and traceability. The first is the classic distinction between functional and non-functional requirements, widely used in both academia and industry. The second is a domain-specific taxonomy aimed at enhancing traceability by organizing requirements through context-aware concepts.

Other classification frameworks—such as BNB (Behavioral/Non-Behavioral), RDAD, FS, and PD—are not detailed here due to scope limitations. Readers interested in a broader theoretical landscape can refer to the comprehensive work by Professor Chikh [6].

## II.3.1 Taxonomy

According to the *Oxford English Dictionary*, taxonomy is defined as:

> "The branch of science concerned with classification, especially of organisms; systematics."

Although taxonomy has its origins in the biological sciences, its core principles have been adapted to other disciplines, including RE. In the context of RE, taxonomy refers to a structured classification system that organizes requirements, processes, artifacts, or concepts to facilitate understanding, analysis, and communication.

This structured approach supports the systematic categorization of different types of requirements (e.g., functional vs. non-functional), stakeholder roles, and methodological steps, thereby improving clarity and consistency in the requirements process. [22]

## II.3.2 Common Types of Taxonomies Relevant to RE

Taxonomies in Requirements Engineering serve to structure knowledge, facilitate communication among stakeholders, and enhance the overall quality of the software development process. Below are two commonly used classifications:

1. **Taxonomies of Requirement Types**

   *Definition and Purpose:* These taxonomies classify requirements into functional, non-functional, domain, and user requirements [26, 7]. Functional requirements define system behaviors; non-functional ones address quality attributes. Domain and user requirements focus on specific contexts and stakeholder expectations [33].

   *Benefits:* They improve requirement coverage, reduce ambiguity, and enhance prioritization and traceability [19]. This structured approach enables development teams to better align requirements with system objectives and stakeholder needs.

   *Challenges:* Differentiating between types can be subjective. Domain expertise and clear stakeholder communication are essential to avoid overlap and vagueness [22]. In practice, requirements often blend categories, which may require iterative clarification and validation.

   *Applications and Tools:* Tools like JIRA or IBM DOORS support such classifications, while standards like IEEE 830 and ISO 29148 help maintain consistency and ensure that requirements are properly categorized and documented [14].

2. **Domain-Specific Taxonomies for Traceability**

   *Definition and Purpose:* These taxonomies are tailored to specific domains, capturing unique concepts and relationships. They enhance traceability by linking requirements to design, implementation, and verification artifacts [29].

   *Benefits:* They support early traceability and align requirements with domain knowledge, which improves the consistency and relevance of downstream development activities [4].

   *Challenges:*

- Defining the right level of granularity [25]
- Non-standardized terminology across domains
- Difficulty in maintenance and ownership [4]
- Ambiguity in classification categories

Furthermore, integrating such taxonomies into fast-paced or cross-disciplinary development settings can be difficult, particularly when domain knowledge is fragmented across stakeholders.

*Applications and Tools:* Common in regulated or complex domains (e.g., healthcare, finance), they are supported by tools like IBM DOORS with domain-specific extensions and ontology-based frameworks [8, 3]. Research prototypes and case studies also continue to refine their practical integration.

In summary, domain-specific taxonomies are essential for improving traceability in software development, offering structured approaches to managing requirements that align with specific domain needs.

## II.4 Conclusion

This chapter has established the foundational concepts and classifications that underpin the practice of Requirements Engineering. By exploring key definitions, taxonomies, and methodological challenges, it provides the theoretical grounding necessary to understand the contributions of the SMART method. The next chapter introduces this method in detail and presents the system designed to operationalize it within a practical software environment.

# III / SMART approach

## CONTENT

This chapter presents the SMART method and its components, based primarily on the work of Professor Azeddine Chikh [6].

# III.1 Introduction

The SMART (Systemic Approach of caTegorizing and Modeling Requirements) method is a novel and systematic approach that contributes to the advancement of Requirements Engineering (RE). It focuses on categorizing and modeling requirements using a well-defined structure. Unlike classical methods, SMART offers a more systematic and scalable solution, which overcomes limitations in addressing complex and evolving challenges. We believe this method has the potential to bring a significant transformation to the field.

To fully appreciate its contribution, it is important to understand the underlying motivations that led to its development. The challenges identified in classical Requirements Engineering — including ambiguity, lack of structure, and evolving stakeholder needs — call for a more adaptive and formalized approach. The SMART method was conceived precisely as a response to these persistent limitations. Its goal is not to replace existing methodologies, but to enhance them by introducing a more refined, structured, and scalable framework for requirement modeling. In doing so, SMART offers a promising direction for tackling the increasing complexity of modern software systems.

One should recognize that this method did not emerge in isolation. As Isaac Newton once said, “If I have seen further, it is by standing on the shoulders of giants.” In the same spirit, SMART builds upon a foundation of well-established models and theories. It incorporates and extends the Requirements Metadata Language (RML) proposed by Chikh (2017) [5]. RML formalizes the qualification of requirements through metadata. SMART also integrates the ISO/IEC 25010 [13] quality model. This model offers a comprehensive view of software characteristics such as usability, reliability, performance, and maintainability.

From Lauesen’s (2002) [17] Goal–Domain–Product scale, SMART derives the BS2 (Business–System–Software) scale. This scale helps distinguish strategic, system-level, and technical concerns. Similarly, the UA (User–Analyst) scale is adapted from the classification proposed by Somerville (2016)[26]. It distinguishes informal user perspectives from formal analyst representations. While these models each provide critical insights, SMART reinterprets and synthesizes them into a unified framework. It also introduces a second dimension that accounts for quality and compliance factors, resulting in a hybrid structure that is both expressive and adaptable.

Beyond its conceptual contribution, SMART introduces the R2F Framework and a hybrid categorization model. These provide a two-dimensional structure for defining, analyzing, and classifying requirements in a reusable and operational way. By integrating taxonomies and scales with metadata-driven traceability, SMART enhances the elicitation, validation, and reuse of requirements while supporting clear communication among stakeholders.

In this way, SMART offers both theoretical innovation and practical value.

It bridges conceptual soundness with operational applicability, making it a compelling method for addressing the increasing complexity and demands of modern software systems.

# III.2 R2F Framework

At the heart of the SMART approach lies the R2F Framework — short for Referential Requirement Framework. This framework offers a structured way to understand and classify software requirements. It was created in response to a key problem in software engineering: the lack of a common, logical structure to describe and analyze requirements. Without such a structure, requirements are often vague, inconsistent, or disconnected from the broader system they belong to. R2F was developed to solve this issue.

R2F provides a two-dimensional reference model for organizing any type of requirement.

## III.2.1 Subjects of Requirements

The first dimension of R2F focuses on the subjects of requirements — that is, the parts of the system or its environments to which the requirement applies. These are grouped into three main categories:

### a) Software:

This category refers to the software product itself, including its functionalities, data, user interfaces, and technical interfaces. These elements form the internal structure of what the software must do and how it interacts with other systems.

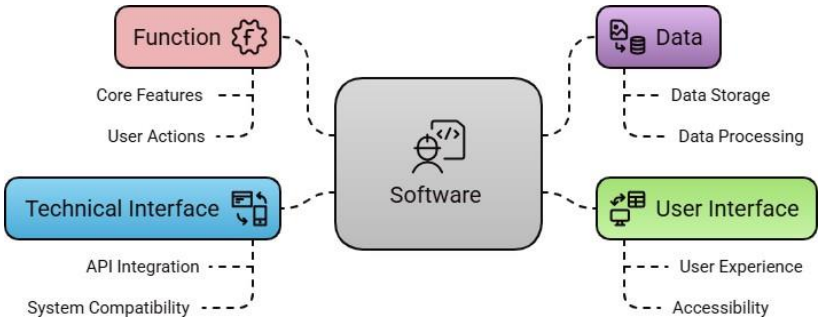



Figure 3.1: Hierarchical structure of the *Software* subject

As shown in figure we have:

- **Function:** Defines what the system must do — its operations and behaviors that fulfill user needs.
  *Example: In a hotel booking app, the system must allow users to filter available rooms by date and price.*

- **Data:** Refers to the structured information the system handles, such as what it stores, retrieves, or processes.
  *Example: In a student management system, each student record must include name, ID, program, and grades.*
- **User Interface:** Describes how users interact with the system — screens, buttons, forms, navigation, etc.
  *Example: In a budgeting mobile app, users must be able to add expenses using a simple input form and see a pie chart of spending.*
- **Technical Interface:** Refers to the system's interactions with external systems or components (e.g., APIs, hardware).
  *Example: In a weather application, the system must fetch current temperature data from an external weather API.*

### b) Operational Environment:

This subject refers to the environment in which the future software will be used. According to the SMART method, it is structured in two levels: five main (generic) subjects at Level 1, each of which may include specific (atomic) subjects at Level 2.

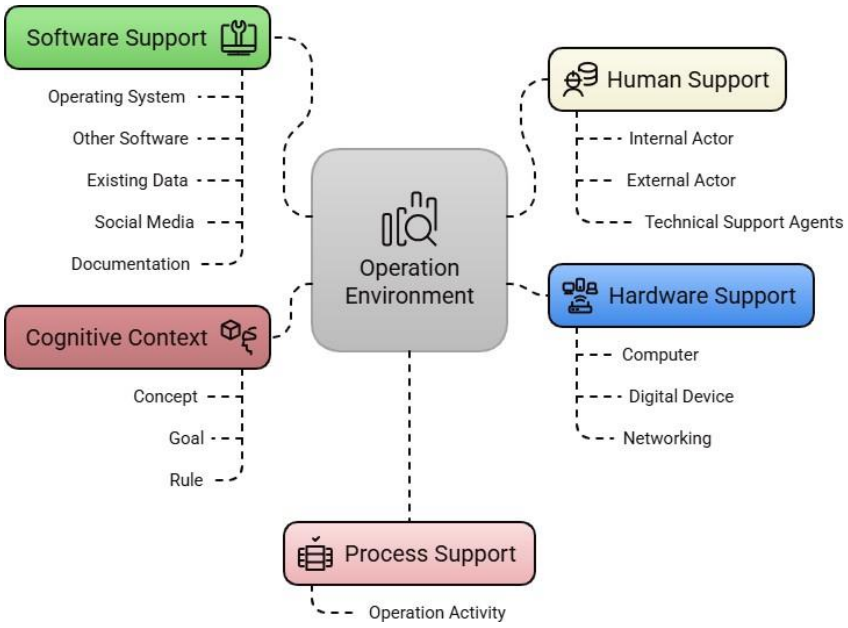



Figure 3.2: Hierarchical structure of the *Operational Environment* subject

- **Process Support for Operation:** Describes the activities that the software must support during its operation.
  - **Operation Activity:** Refers to operational or decision-making tasks the software must support, assist, or automate.
    *Example: In an e-commerce system, the software must support the order delivery and payment confirmation process.*
- **Human Support for Operation:** Identifies the users of the system.

- **Internal Actor:** Employees or users within the organization.
  *Example: Warehouse staff using an inventory system to update stock levels.*
- **External Actor:** Customers or partners interacting with the system.
  *Example: A customer placing an order through a food delivery app.*
- **Technical Support Agent:** Individuals (internal or external) responsible for maintaining or assisting with the system.
  *Example: A helpdesk technician diagnosing system errors remotely.*

- **Software Support for Operation:** Represents the external software and data environments needed.
  - **Operating System:** The OS where the software runs.
    *Example: The app must run on Android 12 and iOS 15.*
  - **Other Software:** Any required external applications.
    *Example: Integration with WhatsApp for customer support.*
  - **Existing Data:** Previously stored or legacy data to be reused.
    *Example: Importing customer records from an old CRM system.*
  - **Social Media:** Digital channels for communication and feedback.
    *Example: Sharing order confirmation via Twitter or email.*
  - **Documentation:** Guides or manuals needed for effective use.
    *Example: An online user manual for system administrators.*

- **Hardware Support for Operation:** Covers physical infrastructure.
  - **Computer:** Machines the software runs on.
    *Example: A laptop used by an employee to access the HR portal.*
  - **Digital Device:** Sensors, tablets, or smart devices.
    *Example: Barcode scanners used in warehouse logistics.*
  - **Networking:** Communication infrastructure.
    *Example: Routers and modems used to ensure system access.*

- **Cognitive Context for Operation:** Describes the knowledge structure behind the domain.
  - **Concept:** Domain-specific terminology or structures.
    *Example: In a university system, concepts like credits, semesters, and GPA.*
  - **Goal:** The user's intended outcomes.
    *Example: A customer's goal to receive an order within 30 minutes.*
  - **Rule:** Business or operational rules the system must enforce.
    *Example: An e-commerce platform must block purchases over €1000 without identity verification.*

### c) Development Environment:

This subject refers to the environment in which the future software will be developed. According to the SMART method, it is organized in two levels: five generic subjects at Level 1, each of which includes specific subjects at Level 2.

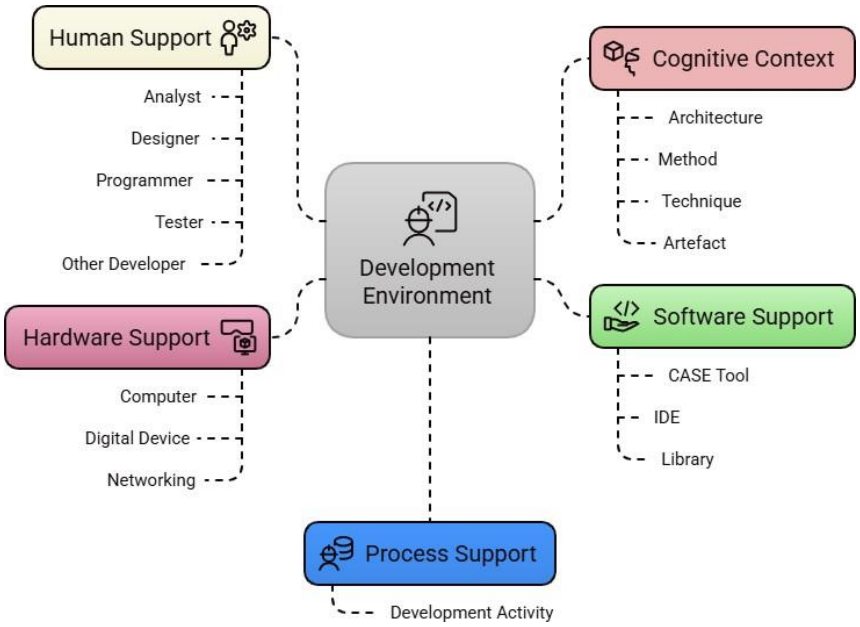



Figure 3.3: Hierarchical structure of the *Development Environment* subject

- **Process Support for Development:** Describes the software engineering activities the system must support during its development.
  - **Development Activity:** Covers all stages such as requirement elicitation, design, implementation, and testing.
    *Example: The system must support code generation after validating design diagrams.*
- **Human Support for Development:** Identifies the people involved in development.
  - **Analyst:** Responsible for requirement elicitation and specification.
    *Example: Analysts define business rules to guide system behavior.*
  - **Designer:** Responsible for both global and detailed design.
    *Example: Designers define system architecture and UI mockups.*
  - **Programmer:** Implements the software in code.
    *Example: Developers write modules based on database specifications.*
  - **Tester:** Ensures the software meets requirements through testing.
    *Example: Testers validate functionalities using test cases.*

- **Other Developer:** Any other contributor to the software development
  *Example: DevOps engineers configure CI/CD pipelines.*

- **Software Support for Development:** Describes the tools used during development.
  - **CASE Tool:** Computer-Aided Software Engineering tool.
    *Example: Use of Visual Paradigm for modeling.*
  - **IDE:** Integrated Development Environment.
    *Example: Using IntelliJ or Visual Studio Code for coding.*
  - **Library:** Software libraries and reusable components.
    *Example: Including a JavaScript chart library like Chart.js.*

- **Hardware Support for Development:** Describes physical infrastructure needed for development.
  - **Computer:** The machine(s) where development occurs.
    *Example: A developer laptop with 16 GB RAM and SSD.*
  - **Digital Device:** Any other smart device used during development.
    *Example: Sensors used in testing an IoT system.*
  - **Networking:** Infrastructure enabling connectivity.
    *Example: A router and secure VPN for remote collaboration.*

- **Cognitive Context for Development:** Defines the methodological and conceptual framework.
  - **Architecture:** The structure of the software to be developed.
    *Example: Adopting a three-tier architecture (presentation, logic, data).*
  - **Method:** The software development method used.
    *Example: Using Agile Scrum methodology.*
  - **Technique:** Specific techniques applied during development.
    *Example: Using prototyping for rapid UI feedback.*
  - **Artefact:** Documents and models produced during development.
    *Example: A sequence diagram created for the login process.*

### III.2.2 Factors of Requirements

The second dimension of R2F concerns the factors that influence requirements. These are contingency factors — conditions or constraints that affect how the requirements should be satisfied. They are grouped into two main categories:

**a) Quality:**

- **Quality Factors:** These factors relate to the desired properties of the future software product and are inspired by the ISO/IEC 25010 standard.

  The SMART method adopts the following eight core factors:

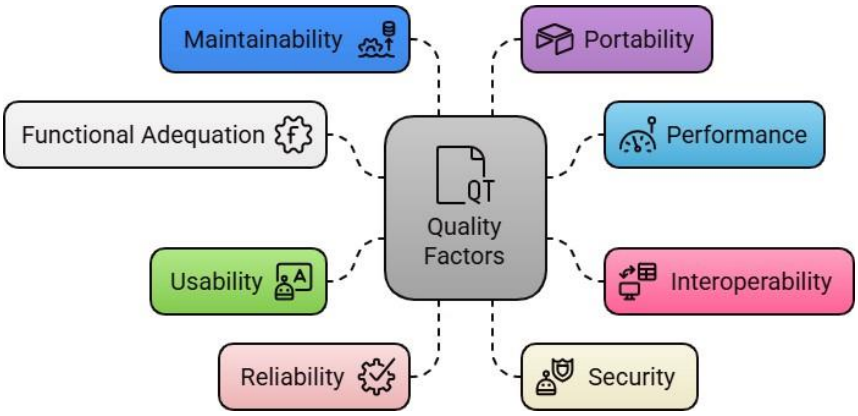



Figure 3.4: Hierarchical structure of the *Quality* factor

- **Functional Adequation:** Measures how well the software meets its functional objectives.
  *Example: A system that allows partial refunds when the order is canceled by the user.*
- **Performance:** Refers to speed, responsiveness, and efficiency.
  *Example: The software must return search results in under 2 seconds.*
- **Interoperability:** Describes the ability of the software to work with other systems.
  *Example: The system must exchange data with a third-party shipping provider.*
- **Usability:** Concerns ease of use, including learnability and accessibility.
  *Example: The app must be usable by people with visual impairments.*
- **Reliability:** Focuses on fault tolerance and consistency.
  *Example: The system must recover automatically after a network disruption.*
- **Security:** Protects against unauthorized access and data breaches.
  *Example: Passwords must be stored using salted hashing.*
- **Maintainability:** Describes how easily the software can be updated or fixed.
  *Example: The code must follow a modular structure to allow easy updates.*
- **Portability:** Indicates the software's ability to operate in different environments.
  *Example: The software must run on both Linux and Windows servers.*

**b) Compliance :**

- **Compliance Factors:** These factors reflect external constraints and environmental conditions that must be respected. The SMART method identifies the following categories:

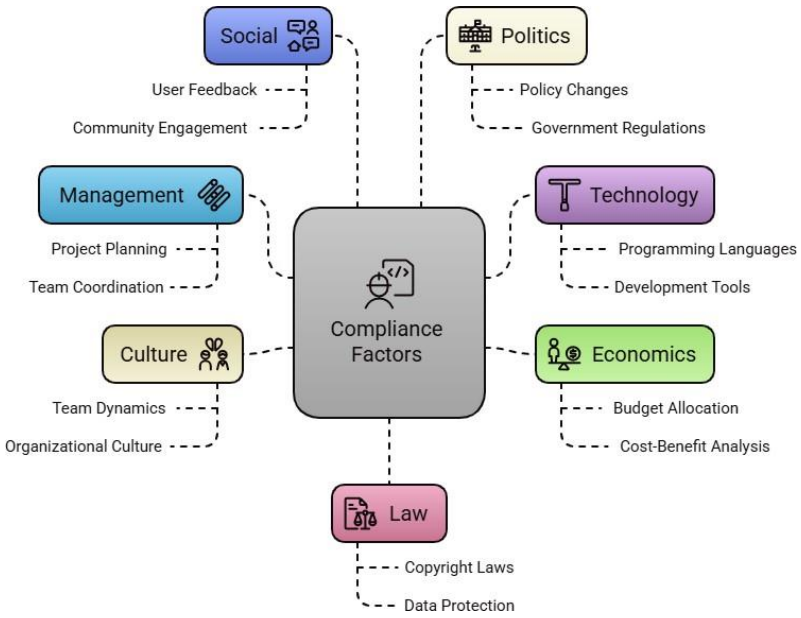



Figure 3.5: Hierarchical structure of the *Compliance* factor

- **Management:** Refers to organizational processes and lifecycle policies.
  *Example: The software must follow internal approval workflows.*
- **Technology:** Encompasses technological standards and expectations.
  *Example: The system must use a specific cloud provider approved by the client.*
- **Economics:** Involves financial limitations or incentives.
  *Example: The total development cost must not exceed $50,000.*
- **Culture:** Includes norms, beliefs, and conventions.
  *Example: The user interface must use date and time formats appropriate for the region.*
- **Social:** Reflects societal behaviors, collaboration needs, and user expectations.
  *Example: The app must include social sharing options to increase engagement.*
- **Politics:** Addresses governmental and organizational power dynamics.
  *Example: The software must comply with state data sovereignty policies.*
- **Law:** Includes legal obligations and standards.
  *Example: The system must comply with GDPR for data privacy.*

## Conclusion

The R2F Framework serves as the structural backbone of the SMART method, providing a clear, extensible model for organizing requirements. Its two-dimensional approach helps ensure consistency, traceability, and relevance across diverse software contexts.*While the figures and classifications were created independently for this work, they were inspired by the structure and concepts introduced in Chikh's SMART method [6].*

# III.3 Hybrid Categorization Approach

## III.3.1 Introduction

Building on the Referential Requirement Framework (R2F), which defines the main subjects and influencing factors in software requirements, this section introduces the hybrid categorization approach proposed by the SMART method. Hybrid categorization refers to a multidimensional classification scheme that combines three perspectives: the origin of the requirement (user or analyst), its level of abstraction (business, system, or software), and its nature (endogenous or exogenous). Unlike traditional single-axis taxonomies, this approach enables a more structured and contextual understanding of requirements.

## III.3.2 Categorization Axes

To ensure a clear and multidimensional classification of requirements, the hybrid categorization approach presented in SMART combines three main axes. Each axis represents a different dimension of the requirement's source, scope, or nature. Together, these axes provide a structured, flexible, and context-aware way to organize requirements.

### UA Scale: User vs Analyst Perspective

The UA (User–Analyst) scale is used to classify requirements based on their origin. This distinction helps clarify the level of formality and structure expected from each type of requirement, as well as the role of the actor expressing it.

- **User-level requirements** are usually expressed in natural language by stakeholders or end-users. These requirements are often informal, broad, and not necessarily structured according to the system's internal logic. They reflect what the user wants to achieve, without specifying how.
- **Analyst-level requirements** are formalized by system analysts. They refine, interpret, and structure the user's needs into precise, testable, and system-compatible specifications. These are often represented using models or structured templates and are directly linked to implementation.

To summarize the distinction:

| Aspect | User | Analyst |
|---|---|---|
| **Author** | Stakeholder | Analyst |
| **Language** | Informal (natural) | Structured/formal |
| **Focus** | Needs/goals | System specs |
| **Level** | High-level | Detailed |
| **Usage** | Initial idea | Design input |
| **Traceability** | Low | High |

Table 3.1: User vs Analyst Requirements – UA Scale

**BS2 Scale: Business, System, Software**

The BS2 scale organizes requirements into three abstraction levels:

- **Business-level:** Focuses on strategic goals and value for the organization.
- **System-level:** Describes what the system must do to support business processes.
- **Software-level:** Specifies how the software should behave or be designed.

| Aspect | Business Level | System Level | Software Level |
|---|---|---|---|
| **Focus** | Organizational goals and benefits | Domain-specific processes and operations | Detailed software behavior and features |
| **Scope** | Strategic and high-level | Functional and operational | Technical and implementation-specifi c |
| **Detail** | Abstract, goal-driven | Medium abstraction, what the system should do | Highly detailed, how the software must act |
| **Examples** | Increase customer satisfaction, reduce cost | Manage inventory, approve requests | Validate form fields, store user input |

Table 3.2: BS2 Scale – Comparison of Business, System, and Software Requirements

### E2 Taxonomy: Endogenous vs Exogenous Requirements

The E2 taxonomy classifies requirements based on their nature and how they relate to the system:

- **Endogenous requirements** describe the internal structure, behavior, and components of the system or its environment. They define what the system *contains or does*.
- **Exogenous requirements** represent external constraints or conditions that the system must comply with. They define how the system *should perform or adapt* in relation to quality attributes or environmental factors.

| Aspect | Endogenous Requirements | Exogenous Requirements |
|---|---|---|
| **Nature** | Internal elements of the system or its environments | External factors influencing the system |
| **Focus** | Functions, data, processes, actors, platforms | Quality attributes (e.g., performance) and compliance (e.g., laws) |
| **Relation to R2F** | Describes *subjects* (e.g., Software, Operation Environment) | Applies *factors* (Quality, Compliance) to subjects |
| **Examples** | User interface, software modules, process flows | Usability, legal compliance, economic constraints |

Table 3.3: E2 Taxonomy – Comparison of Endogenous vs Exogenous Requirements

### III.3.3 Cross-Combination and Final Hybrid Model

To address the complexity and diversity of requirements in system engineering, the SMART method introduces a hybrid categorization model based on the cross-combination of three axes: the author perspective (User vs Analyst), the abstraction level (Business, System, Software — BS2), and the requirement nature (Endogenous vs Exogenous — E2). This structured framework enables a comprehensive view of how requirements originate, how abstract or concrete they are, and whether they describe internal functionality or external constraints.

The purpose of this cross-combination is to facilitate the traceability, refinement, and validation of requirements throughout the development lifecycle. It serves as a bridge between high-level user needs and low-level technical specifications by mapping unstructured expressions into analyzable categories. The following categorization tables reflect the result of this hybrid model, distinguishing between informal user requirements and structured analyst-driven requirements according to their BS2 and E2 attributes.

| User Requirements – Unstructured, not directly linked to R2F | |
|---|---|
| **A — Software-Level Requirements** | |
| **Endogenous** | Informal expressions of software behavior as stated by users.<br>• Desires for features or tools<br>• Expectations for actions or flows<br>• Interface behavior ideas |
| **Exogenous** | User expectations regarding quality, usability, or constraints.<br>• "It should be fast"<br>• "It must be secure"<br>• "It should comply with regulations" |
| **B — System-Level Requirements** | |
| **Endogenous** | User-reported needs related to how the system performs its core operations.<br>• Workflow expectations<br>• Role-based behavior<br>• Support for tasks or routines |
| **Exogenous** | External factors influencing how the system must behave (from a user's view).<br>• Simplicity, responsiveness<br>• Support for regulations<br>• Cultural constraints (e.g., language, layout) |
| **C — Business-Level Requirements** | |
| **Endogenous** | High-level needs or goals expressed by users from a business perspective.<br>• Improve productivity<br>• Reduce errors or costs<br>• Align with business strategy |
| **Exogenous** | External influences on business needs, from the user's viewpoint.<br>• "We need to follow GDPR"<br>• "Must work across regions"<br>• Market or client constraints |



Table 3.4: User Requirement Categorization – Colored by BS2 and E2 Dimensions

| Analyst Requirements – Structured and Referencing R2F | |
|---|---|
| **A — Software-Level Requirements** | |
| **Endogenous** | Defined directly by analysts to describe the core software system behavior.<br>• Functional requirements<br>• Data requirements<br>• UI/interface specifications<br>• Technical integration |
| **Exogenous** | Analyst-specified quality and constraint requirements for the software.<br>• Usability, performance, security<br>• Maintainability, portability<br>• Legal, economic, or social compliance |
| **B — System-Level Requirements** | |
| **Endogenous** | Requirements supporting domain operations and described at the system level.<br>• Operation processes<br>• Human or software support<br>• Cognitive or hardware support |
| **Exogenous** | Constraints or qualities affecting system performance and behavior in use.<br>• Real-world performance<br>• Usability in operations<br>• Legal or cultural restrictions |
| **C — Business-Level Requirements** | |
| **Endogenous** | Strategic internal goals modeled by analysts to align with organizational vision.<br>• Business rules and functions<br>• Structural support for business objectives<br>• Internal goal fulfillment |
| **Exogenous** | External business constraints interpreted and modeled by analysts.<br>• Market/legal compliance<br>• Social or cultural limitations<br>• Ethical and political constraints |



Table 3.5: Analyst Requirement Categorization – Colored by BS2 and E2

### III.3.4 Advantages and Practical Implications of Hybrid Categorization

The hybrid categorization model introduced in the SMART approach enhances the effectiveness of requirements engineering by offering a multidimensional and structured framework. Below are the key advantages:

**1. Multidimensional Structuring and Context Awareness:** By combining the axes of origin (User–Analyst), abstraction level (Business–System–Software), and requirement nature (Endogenous–Exogenous), the model provides a layered view of each requirement. This multidimensional structure allows requirements to be interpreted not only by content, but also by their source, scope, and strategic relevance.

**2. Refinement and Traceability Support:** The distinction between informal user expressions and structured analyst specifications supports a progressive refinement process. This improves traceability from high-level goals to low-level implementation details, ensuring that user intent is maintained throughout the system lifecycle.

**3. Full Requirement Coverage and Consistency:** The hybrid model accounts for both internal functionality and external constraints, covering functional, non-functional, and contextual requirements. This holistic coverage ensures that no relevant aspect of the system or its environment is overlooked.

**4. Reusability and Specification Clarity:** Requirements categorized using this model can be stored, retrieved, and reused across projects with minimal ambiguity. This promotes standardization and leads to clearer, more maintainable specification documents.

**5. Tool Integration and Collaboration Guidance:** The model's structured nature makes it ideal for integration into software tools such as ExpertDASH. It also clarifies stakeholder roles by associating requirement types with users or analysts, which fosters better collaboration and role responsibility during requirements elicitation and validation.

**Practical Implication** In practice, this approach provides a solid foundation for iterative and structured requirement modeling. It improves communication among stakeholders, supports automation, and enables the creation of high-quality, reusable SRS content aligned with both user needs and system goals.

## III.4 Conclusion

This chapter presented the core principles of the SMART method, including the R2F Framework and hybrid categorization model. Together, they provide a structured approach for organizing, classifying, and modeling software requirements. These foundations are essential for developing tools that enable the practical application of SMART in real-world settings. The following chapter builds on this by detailing the implementation of the Expert Dashboard.

# IV / System Analysis

## CONTENT

# IV.1 Introduction

In this section, we describe what the system is expected to do and why.

The system aims to implement the SMART method through a structured web application designed for expert users. We begin by modeling interactions between the system and its users using UML diagrams. We also introduce the system's requirements by categorizing them into functional and non-functional types. This includes:

- The two types of requirement specification: functional requirements -that define what the system does- and Non-functional requirements that define system quality and constraints.
- A use case diagram illustrating the main interactions of the Expert with the system.
- System sequence diagrams detailing key scenarios such as "Add Generic Subject".

These models and specifications offer a comprehensive view of system behavior and form the foundation for further design and implementation.

# IV.2 System Requirements Specification

This section outlines the functional and non-functional requirements of the Expert Dashboard system. These requirements serve as the foundation for system design, use case development, and implementation.

## IV.2.1 Functional Requirements

- **FR1:** The system shall allow authenticated experts to create, modify, and delete Software Requirements Specification (SRS) profiles through a three-step guided interface.
- **FR2:** The system shall enable experts to define, edit, and delete constraints associated with SRS profiles using a dynamic form interface.
- **FR3:** The system shall provide interfaces to manage SMART elements, including Generic Subjects, Generic Factors, Taxonomies, and Scales.
- **FR4:** The system shall allow experts to organize requirements hierarchically using the R2F Framework structure (subjects and factors).
- **FR5:** The system shall generate and store finalized SRS profiles in both JSON and PDF formats.
- **FR6:** The system shall display previously created SRS profiles in a searchable and filterable SRS Library.
- **FR7:** The system shall validate user input before submission, including duplicate checks and hierarchical consistency.

### IV.2.2 Non-Functional Requirements

- **NFR1 – Usability:** The system shall offer an intuitive, responsive user interface that functions seamlessly across common desktop browsers.
- **NFR2 – Performance:** The system shall respond to user interactions (e.g., form submissions, filtering, browsing) within few seconds in 90% of cases.
- **NFR3 – Maintainability:** The system shall follow a modular, three-tier architecture (presentation, logic, and data layers) to facilitate future updates and bug fixes.
- **NFR4 – Security:** All actions shall be accessible only to authenticated expert users; access shall be protected by login and session validation.
- **NFR5 – Traceability:** The system shall support traceability between requirements, classifications, and constraints via structured metadata and semantic links.
- **NFR6 – Reusability:** All SMART model elements (e.g., constraints, taxonomy categories, subjects) shall be reusable across different SRS profiles.
- **NFR7 – Extensibility:** The system shall be designed to support future expansion, including the addition of analyst or administrator roles.
- **NFR8 – Interoperability:** The system shall export structured SRS data in standard formats (JSON, PDF) for use with external tools.

## IV.3 Use Case Diagram

A use case diagram illustrates the interactions between external actors and the system. The diagram below shows the interaction between the **Expert** and the **Expert Dashboard**.

Below is a list of the main use cases identified in the system along with a brief explanation of each:

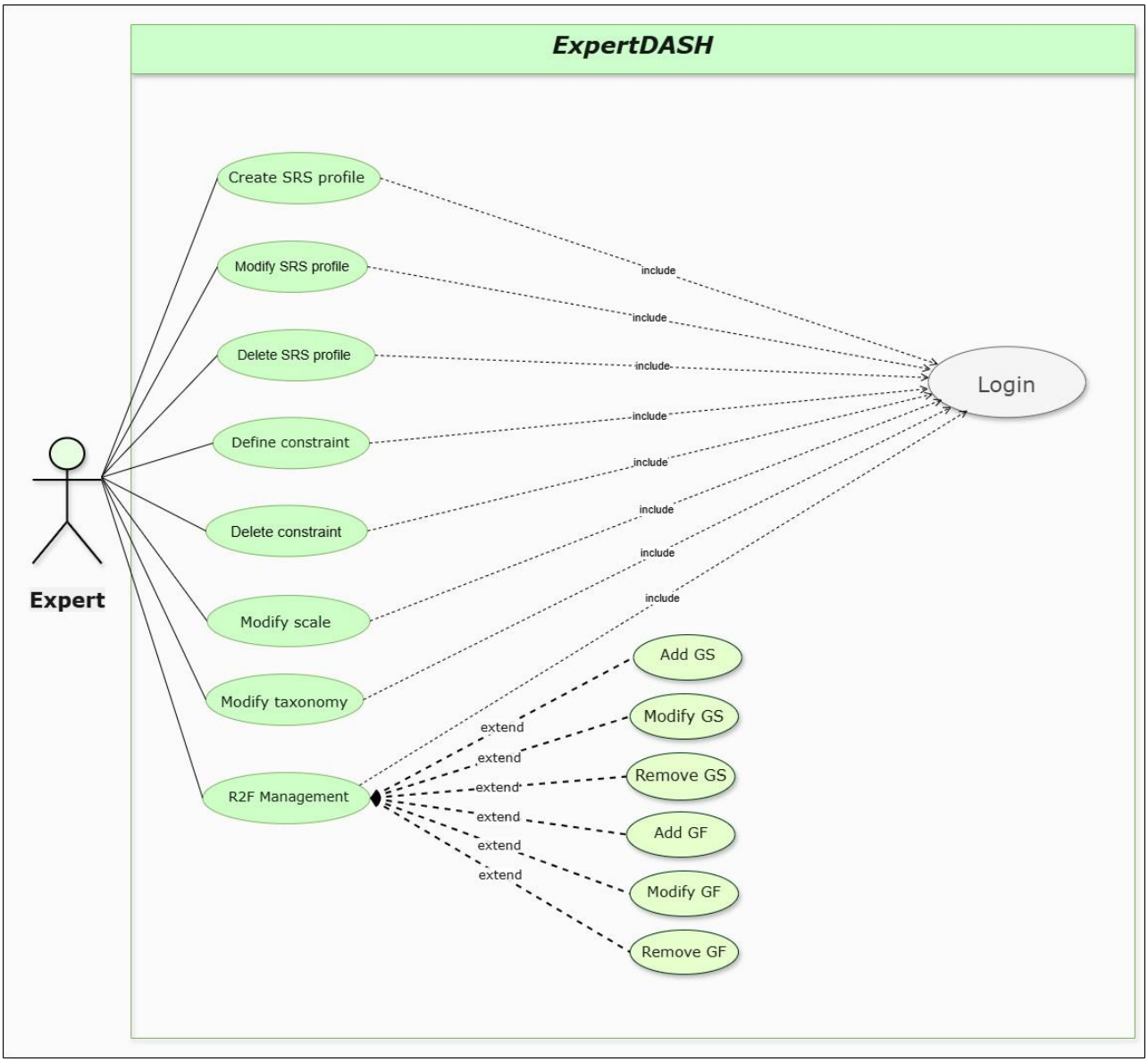



Figure 4.1: Use Case Diagram for Expert Dashboard

## IV.3.1 Use cases:

a) **Create an SRS Profile** Through a guided interface, the expert completes three steps: entering metadata, associating requirements via R2F, and defining constraints. The finalized profile is saved in JSON and PDF formats.

b) **Modify or Delete an SRS Profile** Experts can access the SRS Library to edit existing profiles or permanently delete them.

c) **Manage Constraints** Constraints can be added, edited, or removed during Step 3 of SRS creation. Each constraint includes fields like type, priority, and status, and is managed through dynamic forms.

d) **Manage Generic Subjects and Factors** Experts can create, modify, or delete generic subjects and factors using popup forms in the Data Explorer interface. The system ensures referential integrity for deletion actions.

e) **Manage Taxonomies and Scales** Taxonomies and evaluation scales are editable through the Data Explorer. Users can update names, values, or hierarchies as needed, with changes reflected across the platform.

# IV.4 System Sequence Diagrams

System sequence diagrams (SSDs) are used to model the interactions between the user (actor), the interface, the system's core logic, and the database. These diagrams capture the dynamic behavior of the system, detailing how information flows through various layers during a use case execution. In this section, we focus on one of the key actions in the system: adding a Generic Subject (GS).

**Note** This sequence is representative of similar interactions throughout the system, such as adding Generic factor, or deleting taxonomy/scale/factor/subject/constraint. As most operations follow a uniform request–form–validation–store pattern, only one representative SSD is included for clarity.

## IV.4.1 Add Generic Subject – Sequence Flow

Once the user is authenticated and logged in, they may access the functionality to add a new Generic Subject via the Data Explorer interface, which provides full access to database entities.

When the user decides to add a Generic Subject, the system verifies their permissions then displays the appropriate input form.

Next, the system fetches existing data from the database, including previously defined Generic Subject records, in order to pre-fill related category suggestions and help maintain consistency. During this step, the system also validates the input for missing fields or invalid characters, and performs a duplicate check to identify any similar existing entries.
If a duplicate is detected, the system shows a warning message and prompts the user to confirm or modify the entry. If the input is valid and unique, the system proceeds to perform a final verification to ensure data consistency.
Finally, once the data passes all checks, the new Generic Subject is saved to the database. A success message is displayed.

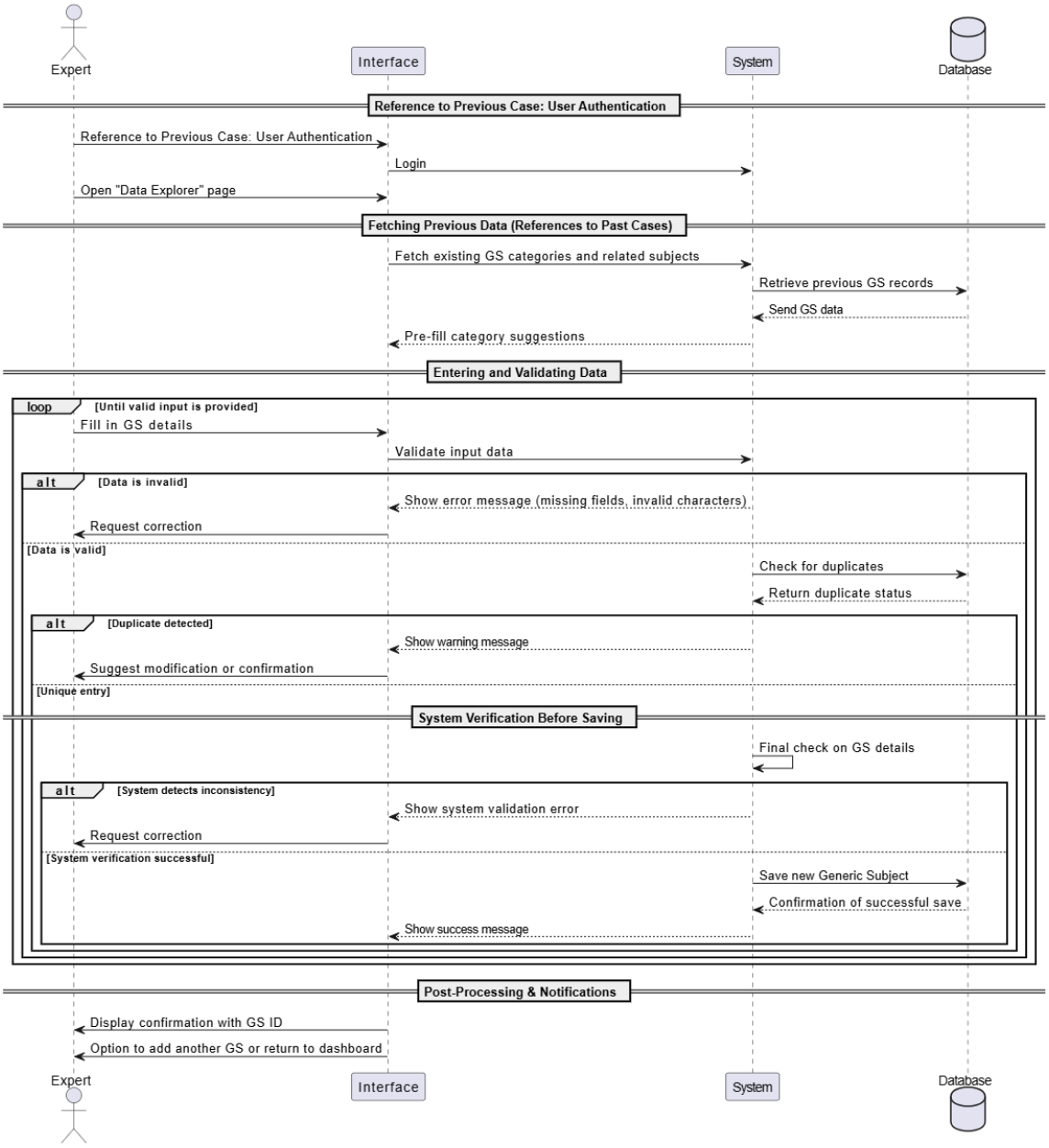



Figure 4.2: System Sequence Diagram for Add Generic Subject

# IV.5 Conclusion

This chapter provided a comprehensive analysis of the system, including its functional and non-functional requirements, core interactions through use case diagrams, and sequence behavior through system sequence diagrams. These foundational elements define the expected functionality and serve as a roadmap for system design.

# V / System Design

## CONTENT

# V.1 Introduction

This chapter presents the design architecture of the Expert Dashboard—an interactive system developed to operationalize the SMART method from the perspective of expert users. As the first implemented module in a broader ecosystem, the Expert Dashboard focuses exclusively on expert-level functionalities: structuring requirements, managing SMART elements such as subjects and factors, and generating Software Requirements Specification (SRS) profiles. The chapter details the overall architecture, key system components, user interface behavior, API communication, state management, and database modeling.

# V.2 General design (System Architecture)

## V.2.1 System Overview

The software system is a web-based application that constitutes the Expert Dashboard: the first developed module of a larger tool intended to support the SMART method (Systemic Approach to Categorizing and Modeling Requirements).

This tool is planned to offer dedicated dashboards for different user roles, including experts, analysts, and administrators. Each dashboard will be tailored to the specific responsibilities and interactions required by its respective user type.

At this stage, the development focuses exclusively on the Expert Dashboard, which enables domain experts to manage requirements, organize elements of the SMART method, and structure specification documents efficiently.

## V.2.2 Architecture Style and Design Rationale

The architecture style used in this system is **three-tier architecture**, chosen for its clear separation of concerns and modularity. This style divides the system into three main layers:

- **Presentation Layer**: This represents the frontend of the application, developed using Vue.js. It is responsible for rendering the user interface and handling user interactions.
- **Application (Logic) Layer**: This layer forms the core of the system's business logic and is implemented using Laravel. It processes requests from the frontend and communicates with the data layer.
- **Data Layer**: This layer is responsible for data storage and retrieval. It is managed using a MySQL database running within a local WAMP environment.

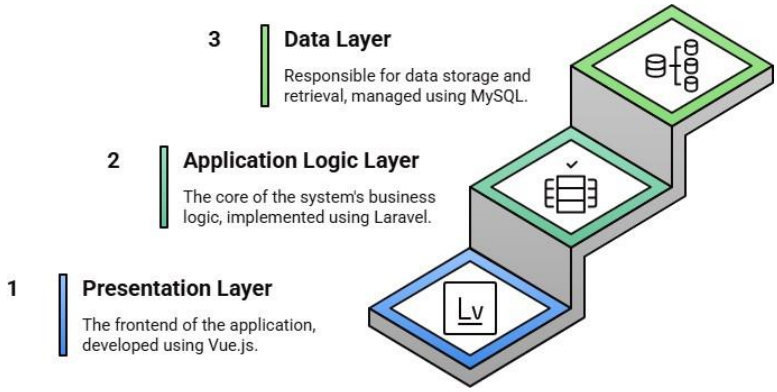



Figure 5.1: Three-Tier Architecture of the Expert Dashboard

### Design Rationale

The selection of technologies for this system was driven by the structural and functional requirements of the SMART method support tool.

- **Database:** The system involves managing hierarchical data structures—such as trees for generic subjects and factors—which made MySQL a suitable choice due to its mature support for relational data, indexing, and efficient querying of structured datasets.
- **Backend:** Laravel was selected for the backend due to its adherence to the Model-View-Controller (MVC) architecture, which promotes clean separation of concerns and maintainable code. Laravel also provides robust built-in features such as routing, middleware, and authentication, all accessible through its intuitive syntax. The framework's Eloquent ORM simplifies database interaction, while its extensive documentation and community support accelerate development. Laravel's built-in tools like Artisan CLI further reduce boilerplate code and streamline common tasks, making it a full-featured and efficient choice for PHP-based backend development.
- **Frontend:** The frontend leverages Vue.js for its structured and component-based approach, which provides a significant improvement over using plain HTML, CSS, and JavaScript. As the first JavaScript framework learned during development, Vue.js offered a smoother transition from traditional frontend technologies while introducing modern development practices such as reactivity, data binding, and reusable components. Although simplicity and accessibility were key factors, Vue.js also proved suitable for this project because of its lightweight nature and ability to manage dynamic interfaces efficiently.

In summary, the chosen architecture and technologies align with the requirements of the SMART method tool, offering an optimal balance between scalability, usability, and maintainability.

## V.2.3 Major System Components

The system is composed of several core components that work together to deliver a functional and modular web application. Each component is designed to handle a specific concern, adhering to the principles of separation of concerns and maintainability.

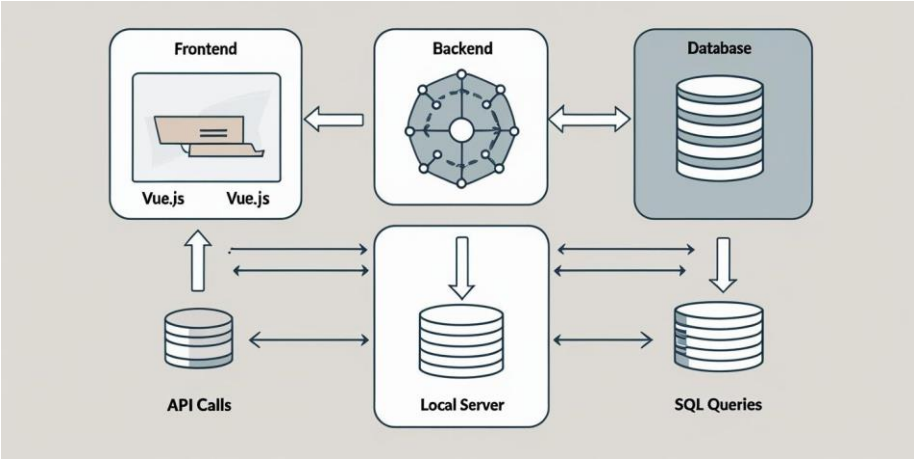



Figure 5.2: Major Components of the Expert Dashboard System

0. **Frontend (Presentation Layer)** The frontend of the application is built using Vue.js, a progressive JavaScript framework. It is responsible for:

   a. Rendering the user interface
   b. Managing user interactions and routing.
   c. Communicating with backend APIs.
   d. Displaying data dynamically through reactive components.

   Vue's component-based structure enables modular development, reusability, and ease of maintenance. It also facilitates two-way data binding and clean integration with backend responses.

0. **Backend (Application Logic Layer)** The backend is developed using Laravel, a PHP framework that follows the Model-View-Controller (MVC) architectural pattern. It provides:

   a. RESTful API endpoints for the frontend
   b. Business logic implementation
   c. Request validation and user authentication
   d. Data processing and manipulation

   Laravel offers tools like Eloquent ORM, middleware, and Artisan CLI, making backend development efficient and organized.

0. **Database (Data Layer)** The system uses MySQL as its relational database management system, hosted on a local WAMP server during development. The database is responsible for:

a. Storing structured data such as users, requirements, SRS profiles, constraints, and method elements (R2F, scales, taxonomy).
b. Managing hierarchical relationships (e.g., between subjects and factors).
c. Ensuring data consistency and integrity through relational constraints and indexing.

0. **Routing and API Communication**

API communication between the frontend and backend is handled via HTTP requests (typically using Axios). Laravel's routing system maps these requests to the appropriate controller actions, returning structured responses (usually JSON) to be processed by the Vue.js frontend.

0. **Local Environment (WAMP Stack)**

During development and testing, the system operates on a WAMP environment (Windows, Apache, MySQL, PHP). This setup provides:

a. A stable local server for development
b. Easy management of the database using phpMyAdmin
c. Compatibility with Laravel and PHP

## V.2.4 Deployment Architecture

The current deployment of the Expert Dashboard module is conducted in a local development environment using the WAMP stack (Windows, Apache, MySQL, PHP). This setup provides an integrated environment for building and testing the system before transitioning to production.

The system architecture follows a **client-server model** with the following deployment structure:

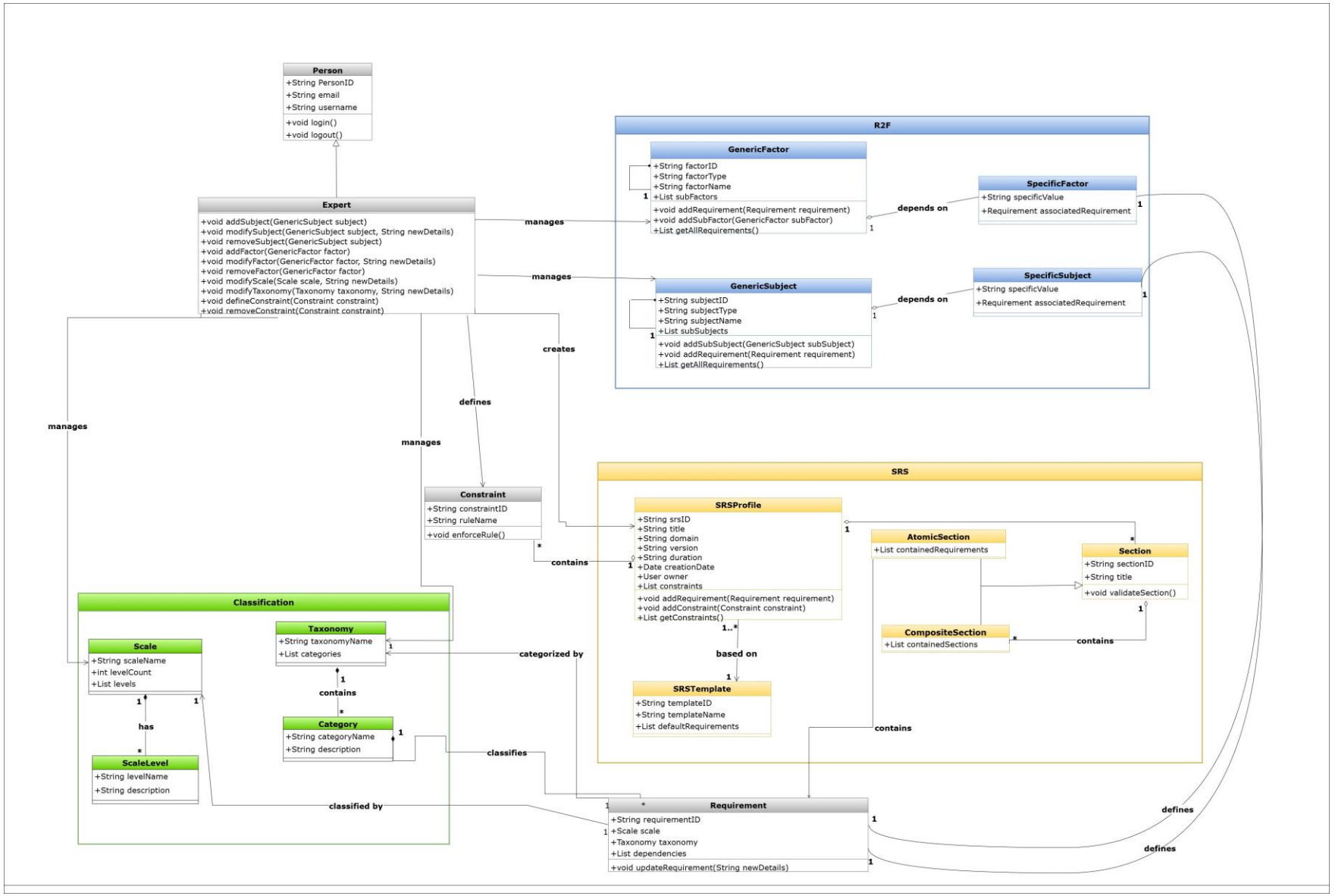
- **Client Side:** The frontend of the system, developed using Vue.js, runs in the user's web browser. It communicates with the backend via HTTP requests.
- **Server Side:** The backend is powered by the Laravel framework, hosted on Apache through the WAMP stack. Laravel handles all business logic, routing, and database interactions.
- **Database:** MySQL, managed through WAMP, stores all system data including user information, SRS documents, R2F elements, constraints, and metadata.

The deployment is currently localized for development purposes, with all components (frontend, backend, and database) running on a single machine (localhost).This simplifies testing and debugging during the initial development phase. This modular setup also facilitates future integration of the AnalystDash, ensuring that the architecture remains extensible as the system evolves.

**Future Deployment Considerations:** Future deployment to a production environment may involve migrating the system to a cloud-based server, separating components across different nodes (e.g., database hosted remotely, backend via API gateway), and implementing advanced features like HTTPS, domain mapping, and user authentication management.

### V.2.5 Class Diagram

The class diagram below represents the structural design of the *Expert Dashboard* system. It models the core entities, their attributes, and the relationships between them, forming the foundation for both the application logic and the underlying database schema.


Person
+String PersonID
+String email
+String username
+void login()
+void logout()
Expert
+void addSubject(GenericSubject subject)
+void modifySubject(GenericSubject subject, String newDetails)
+void removeSubject(GenericSubject subject)
+void addFactor(GenericFactor factor)
+void modifyFactor(GenericFactor factor, String newDetails)
+void removeFactor(GenericFactor factor)
+void modifyScale(Scale scale, String newDetails)
+void modifyTaxonomy(Taxonomy taxonomy, String newDetails)
+void defineConstraint(Constraint constraint)
+void removeConstraint(Constraint constraint)
manages
manages
creates
defines
manages
manages
R2F
GenericFactor
+String factorID
+String factorType
+String factorName
+List subFactors
+void addRequirement(Requirement requirement)
+void addSubFactor(GenericFactor subFactor)
+List getAllRequirements()
depends on
SpecificFactor
+String specificValue
+Requirement associatedRequirement
GenericSubject
+String subjectID
+String subjectType
+String subjectName
+List subSubjects
+void addSubSubject(GenericSubject subSubject)
+void addRequirement(Requirement requirement)
+List getAllRequirements()
depends on
SpecificSubject
+String specificValue
+Requirement associatedRequirement
Constraint
+String constraintID
+String ruleName
+void enforceRule()
contains
SRS
SRSProfile
+String srsID
+String title
+String domain
+String version
+String duration
+Date creationDate
+User owner
+List constraints
+void addRequirement(Requirement requirement)
+void addConstraint(Constraint constraint)
+List getConstraints()
based on
SRSTemplate
+String templateID
+String templateName
+List defaultRequirements
AtomicSection
+List containedRequirements
Section
+String sectionID
+String title
+void validateSection()
CompositeSection
+List containedSections
contains
contains
contains
Classification
Scale
+String scaleName
+int levelCount
+List levels
has
ScaleLevel
+String levelName
+String description
Taxonomy
+String taxonomyName
+List categories
contains
Category
+String categoryName
+String description
categorized by
classifies
classified by
Requirement
+String requirementID
+Scale scale
+Taxonomy taxonomy
+List dependencies
+void updateRequirement(String newDetails)
defines
defines

The diagram is structured into several conceptual domains:

- **R2F Framework:** Captures the abstraction layers of subjects and factors, distinguishing between generic reusable entities and project-specific instances.
- **SRS Profile Structure:** Models the hierarchical structure of an SRS document, using atomic and composite sections along with constraints and requirements.
- **Classification Layer:** Represents the use of SMART method constructs such as taxonomy, scales, and categories to support requirement prioritization and analysis.
- **Expert and Classification Management:** Depicts how the Expert class inherits from Person and manages core classification structures such as Scale, Taxonomy, and Constraint, supporting the SMART-based organization of requirements.

# V.3 Detailed Design

## V.3.1 Application Design

The ExpertDASH platform adopts a modular, service-oriented architecture, designed to separate concerns between the user interface and the underlying business logic. This separation enhances scalability, system clarity, and ease of future integration with external systems.

**System Architecture** At its core, the system follows a client-server model. The backend handles all business logic and data management, exposing its functionalities through RESTful APIs. The frontend communicates with these APIs to fetch and display data, enabling a responsive and interactive user experience.

**Authentication and Security** User sessions are secured through token-based authentication, ensuring that only authorized users can access or modify data. The system implements role-based access control to regulate permissions based on user roles, particularly for sensitive operations like managing system content or editing specifications.

**Modular Functional Components** The application is structured into independent modules, each responsible for a specific aspect of functionality. These include:

- **User and Expert Management:** Supports registration, authentication, and role assignment.
- **SRS Management:** Facilitates creation and editing of Software Requirements Specification profiles.
- **Requirement and Constraint Handling:** Enables experts to define, classify, and associate requirements and constraints.
- **Taxonomy and Scale Management:** Provides interfaces for managing classification schemes and scoring systems.

### Component Communication

The interaction between the frontend and backend follows a structured flow. The user interface initiates actions (such as submitting forms or retrieving data), which are processed by the backend and returned in structured responses. The frontend then updates the displayed content accordingly, ensuring a seamless user experience.

### Routing and Navigation

ExpertDASH is built as a single-page application (SPA), using dynamic routing to manage navigation between different modules. Each module (such as the dashboard, SRS management, or classification system) is linked to a dedicated route, allowing users to navigate without full page reloads.

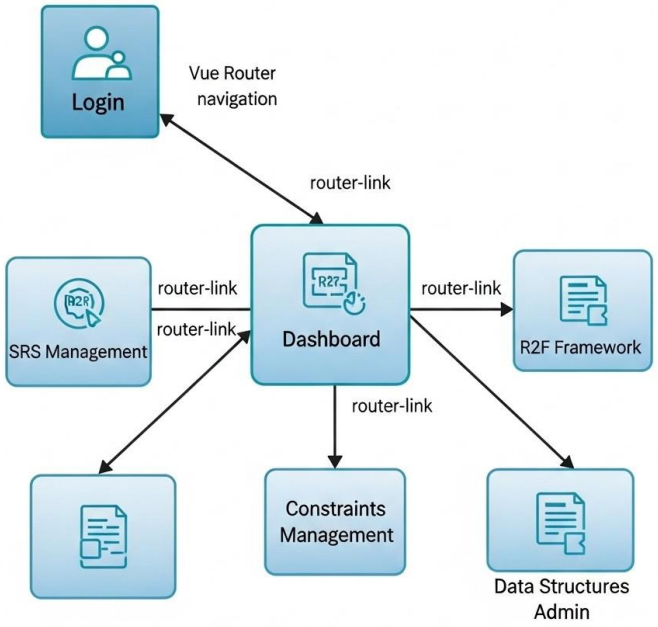



Figure 5.4: Navigation Flow within the ExpertDASH Interface

This design offers several benefits:

- **Modularity and Maintainability:** Each module is encapsulated and independently manageable.
- **Consistent User Experience:** Uniform navigation behavior across modules.
- **Scalability:** New modules can be added with minimal restructuring.

Visual cues such as highlighted menu items guide users through the application, reinforcing navigation intuitiveness and system usability.

## V.3.2 Database Design

### Entity-Relationship Diagram

The database in the ExpertDASH system stores and organizes all the information related to users, experts, requirements, documents, and classifications. The Entity-Relationship (ER) model helps define how these elements are connected and ensures that data remains consistent and reliable.

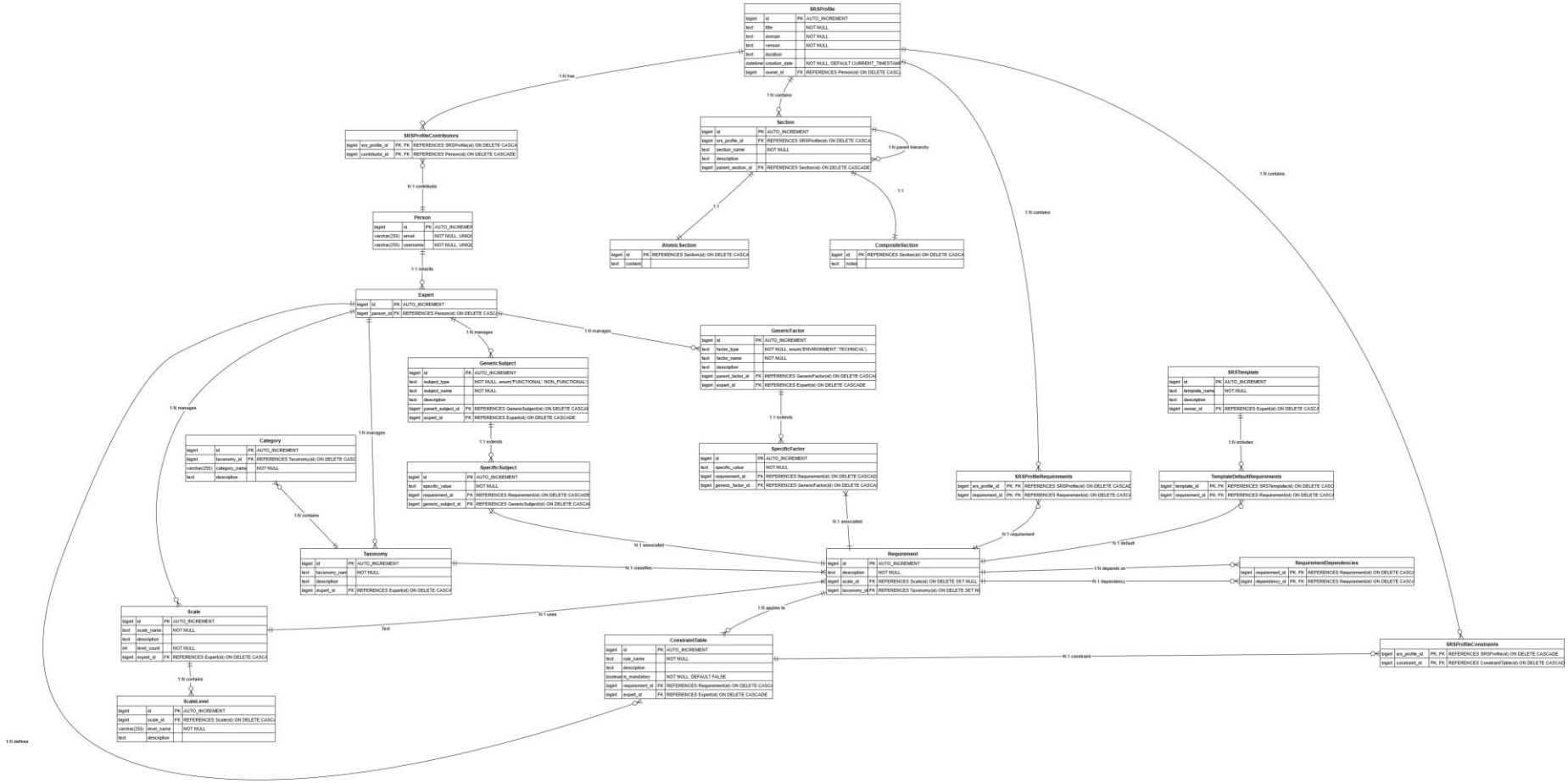



Figure 5.5: Entity-Relationship Diagram of the ExpertDash System

### Main Components of the Schema

The database is divided into multiple logical parts to handle different functionalities:

- **Users and Experts:** These tables manage login information and expert accounts.
- **Requirements and Constraints:** Requirements are central elements of the system, and experts can define constraints to guide how they should be applied.
- **Categorization and Evaluation:** Requirements are organized using taxonomies and categories, and they can be evaluated using scales and levels defined by experts.
- **Document Management:** SRS profiles and templates help structure requirement documents. Sections can be nested to create detailed documents.
- **R2F Framework:** Generic and specific subjects and factors are used to build a hierarchical structure of requirement domains.
- **Supporting Tables:** Additional tables exist to help with login sessions and security but are not directly related to the core features.

### Support for Hierarchical Structures

Some data in the system is organized in a tree-like structure. For example, a section in a document can contain sub-sections, or a subject can have sub-subjects. The database uses a special structure called "self-referencing" to allow this nesting.

This approach makes it easier to:

- Represent complex, layered documents.
- Group related subjects or factors under one parent.
- Maintain consistency when one part is updated or removed.

## V.4 Conclusion

The system design phase provided a structured blueprint for building the Expert SRS platform. By defining the system architecture, data models, and key interface components, this chapter laid the foundation for the implementation stage. The design choices aimed to ensure modularity, scalability, and maintainability, while addressing the functional and non-functional requirements identified during analysis. Overall, the system design offers a clear roadmap for developing a reliable and efficient software solution.

# VI / System Implementation

## CONTENT

# VI.1 Introduction

This chapter presents the implementation of the **ExpertDASH** web platform, which serves as the technical realization of the SMART-based requirements engineering framework introduced in earlier chapters. ExpertDASH is designed to support domain experts in the modeling, categorization, and structured management of software requirements specifications (SRS) through an interactive and scalable web-based interface.

The system architecture was developed based on the conceptual and functional models discussed in the design phase, ensuring full alignment with SMART principles such as Generic Subjects (GS), Generic Factors (GF), and Constraints.

# VI.2 Tools and technologies used


Figure 6.1: Used technologies

## VI.2.1 Laravel:

A free and open-source PHP framework used for backend development, designed for the creation of web applications following a well-known architectural pattern called Model–View–Controller (MVC).It was created by Taylor Otwell and released in June 2011.The framework is regularly updated with new versions.[16]

## VI.2.2 PHP:

A server-side scripting language designed for web development, primarily for creating dynamic websites. It is widely used, cross-platform, and integrates with databases.[20]

## VI.2.3 MySQL:

An open-source relational database management system (RDBMS) used for storing and managing data, with both backend and frontend integration.[18]

### VI.2.4 phpMyAdmin :

A free and open source administration tool for MySQL and MariaDB.Provides an intuitive secure web-based interface for managing databases,it also enables importing, exporting, and backing up data. [21]

### VI.2.5 Postman :

An application for API testing , it uses json format usually and works with endpoints.

Postman supports different HTTP methods ( GET,POST,PUT,DELETE...).[23]

### VI.2.6 Visual Studio Code :

A popular source code editor that supports multiple programming languages and offers advanced functionalities such as integrated debugging, automatic code completion, and Git integration. [30]

### VI.2.7 Vue.JS:

A JavaScript framework used for frontend development. It is component-based that supports reactive data binding to create dynamic web apps. Vue.js focuses mainly on the View layer, is easy to integrate into existing projects [31].

### VI.2.8 Tailwind css :

A modern CSS framework that relies on predefined utility classes from its own library to build custom designs quickly and efficiently. [27]

### VI.2.9 Axios :

An HTTP client service for JavaScript that simplifies frontend communication with backend APIs.It is often used in Vue.js.[1]

### VI.2.10 WAMPServer :

stands for Windows, Apache, MySQL, PHP. it is web devolopement platform that is used by developers because it is easy to run and test PHP-based web applications locally without needing an external server.[32]

### VI.2.11 GitHub :

A hosting platform based on Git, a version control system. It is common amongest developers to share projects by storing them in repositories,collaborate with others,while also track and manage changes made to the code over time. [10]

## VI.3 User Interfaces Overview

This section presents the main user interfaces of the *ExpertDASH* system, highlighting their structure, functionality, and interactions. Each interface is designed to support experts in defining, managing, and organizing software requirements through a guided and intuitive experience.

### VI.3.1 Login Interface

The login interface is the entry point to the *ExpertDASH* platform (Figure 6.2), providing secure access to authorized users. It features a minimalistic design that prompts users to enter their credentials, which are verified via the backend authentication service. Upon successful login, the user is redirected to the main dashboard, with mechanisms in place to ensure the protection of sensitive project and requirement data.

It is important to note that the login interface does not include a public "Create Account" option. This is an intentional design choice aimed at maintaining system integrity and access control. Account creation is restricted to an authorized expert, who can manage user credentials via the dedicated "Accounts" section in the Settings interface (see Figure 6.14). This approach ensures that only verified users are granted access, reducing the risk of unauthorized entry and simplifying user management in expert-driven environments.

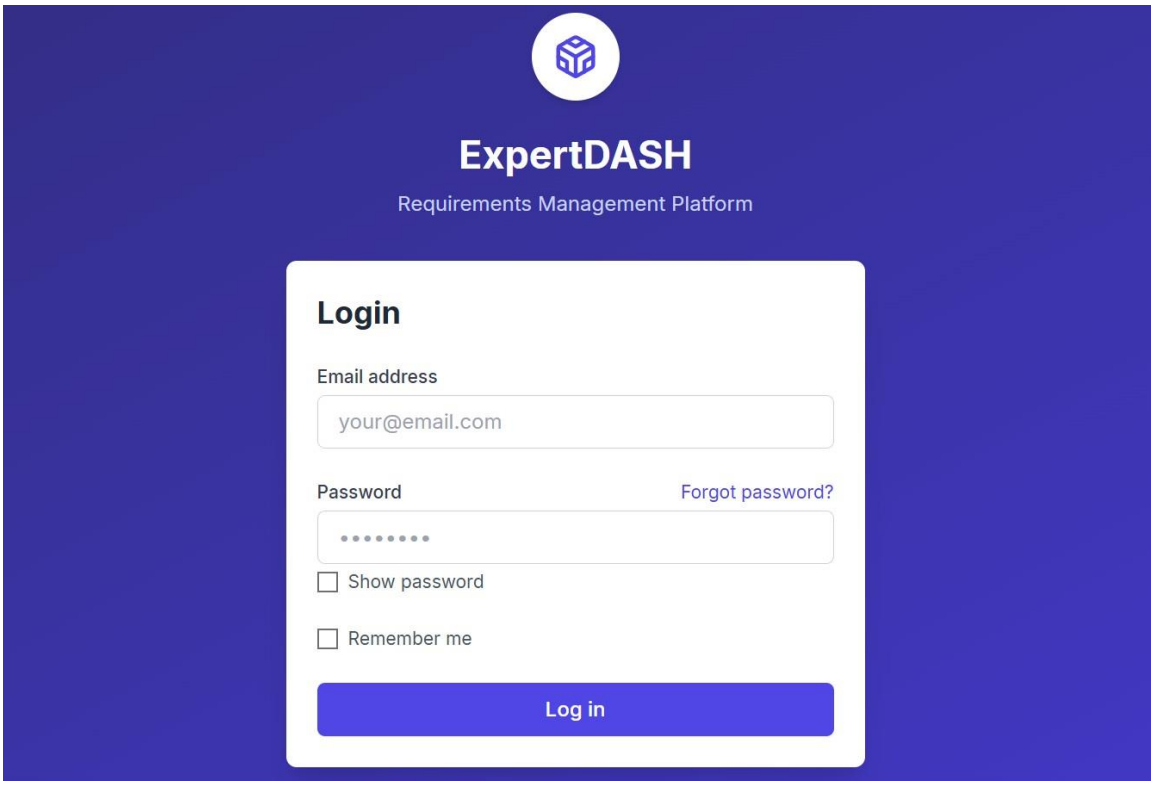



Figure 6.2: ExpertDASH Login Interface

## VI.3.2 Dashboard

The Dashboard (Figure 6.3) serves as the main landing page of the ExpertDASH platform after successful authentication. It provides an at-a-glance overview of the system's core functionalities and recent activities, enabling experts to navigate seamlessly through the SRS management workflow.

It includes shortcut cards, visual cues, and contextual information to help users quickly access modules such as SRS creation, the data explorer, or the SRS library. The dashboard design prioritizes clarity and responsiveness to enhance the user experience across different devices and screen sizes.

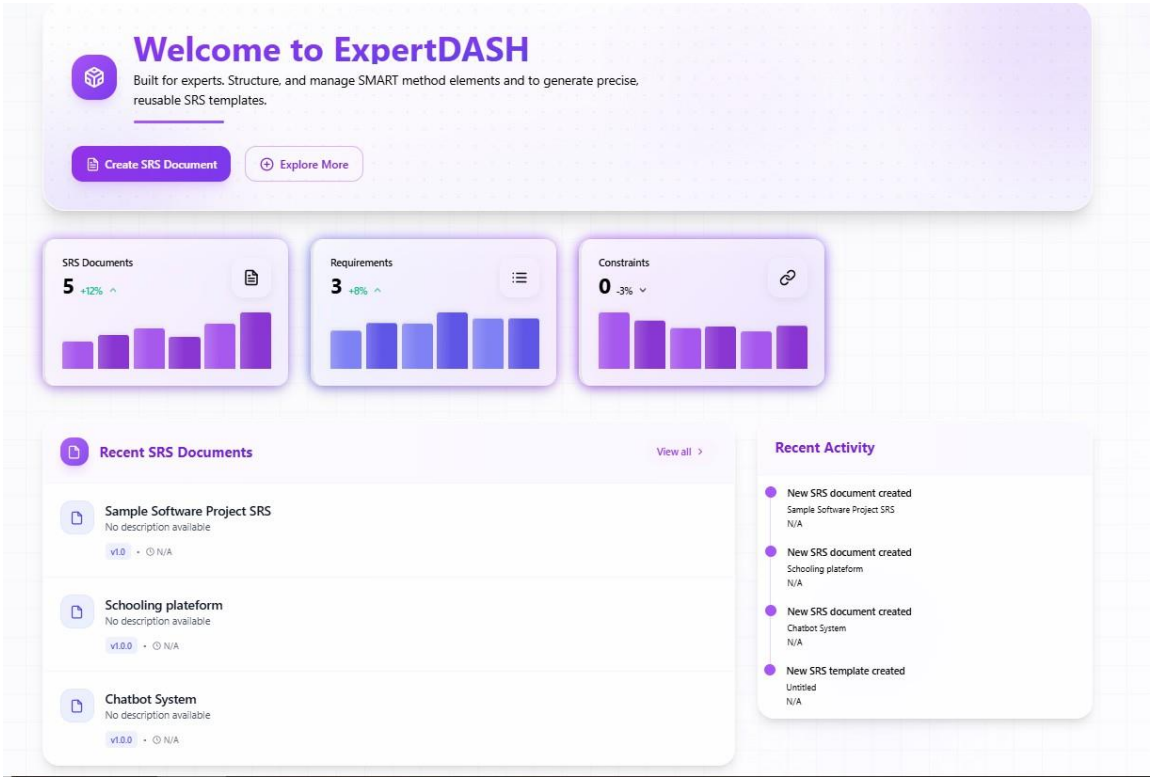



Figure 6.3: Dashboard Interface

## VI.3.3 Unified SRS Creation Interface

The Unified SRS Creation module in ExpertDASH is designed as a guided process to assist experts in building complete and structured Software Requirements Specification (SRS) documents. The process is divided into three main steps: (1) General Information, (2) R2F Framework, and (3) Constraints. Each step presents its own interface for inputting and managing the corresponding requirement elements.

### Step 1 – General Information

In the first step (Figure 6.4), the expert enters basic information about the SRS document, including its title, domain, version number, and expected duration. This data forms the foundation for organizing and referencing the SRS profile.

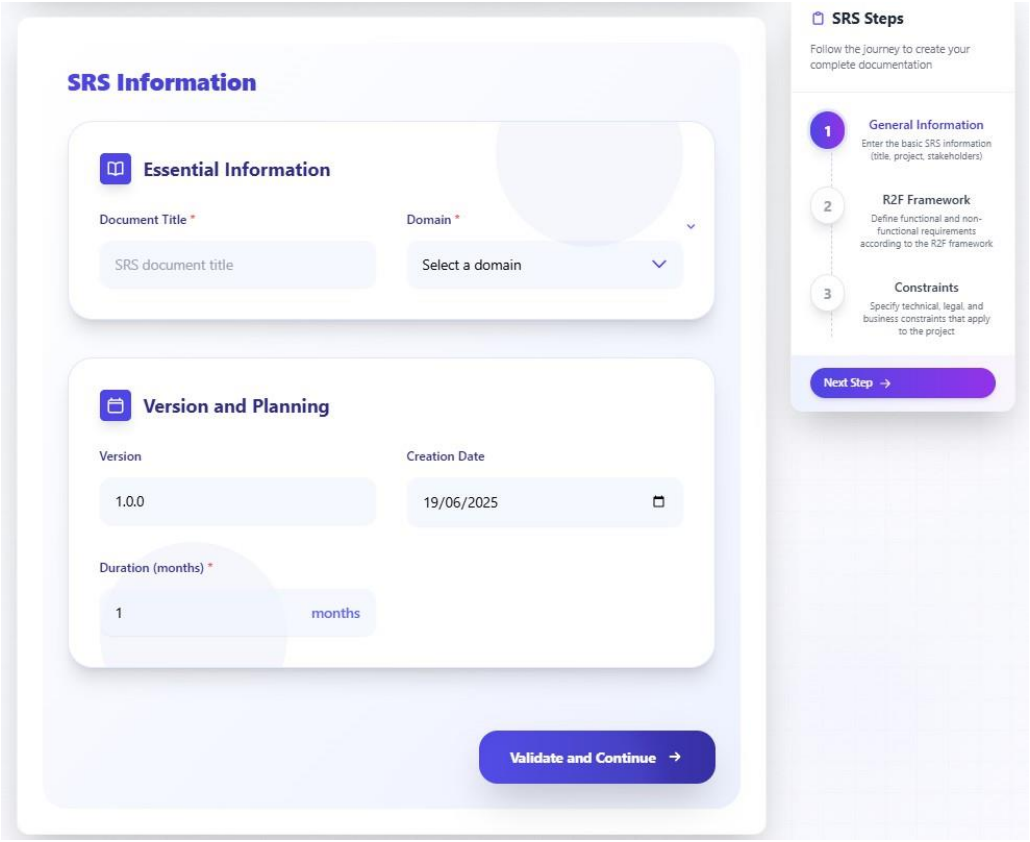



Figure 6.4: Step 1 – General Information input interface

## Step 2 – R2F Framework Selection

The second step (Figure 6.5 and Figure 6.6),allows the expert to define the core content of the SRS document by selecting relevant elements from the R2F Framework. This step includes the selection of Generic Subjects and Generic Factors, each displayed in separate interactive panels that support both browsing and detailed input.

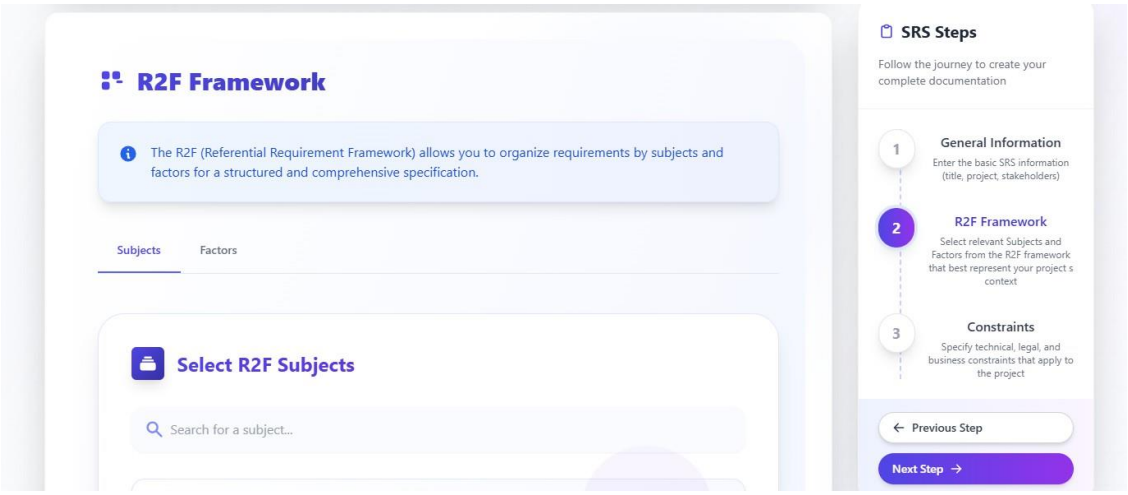



Figure 6.5: Step2: Generic Subjects Selection

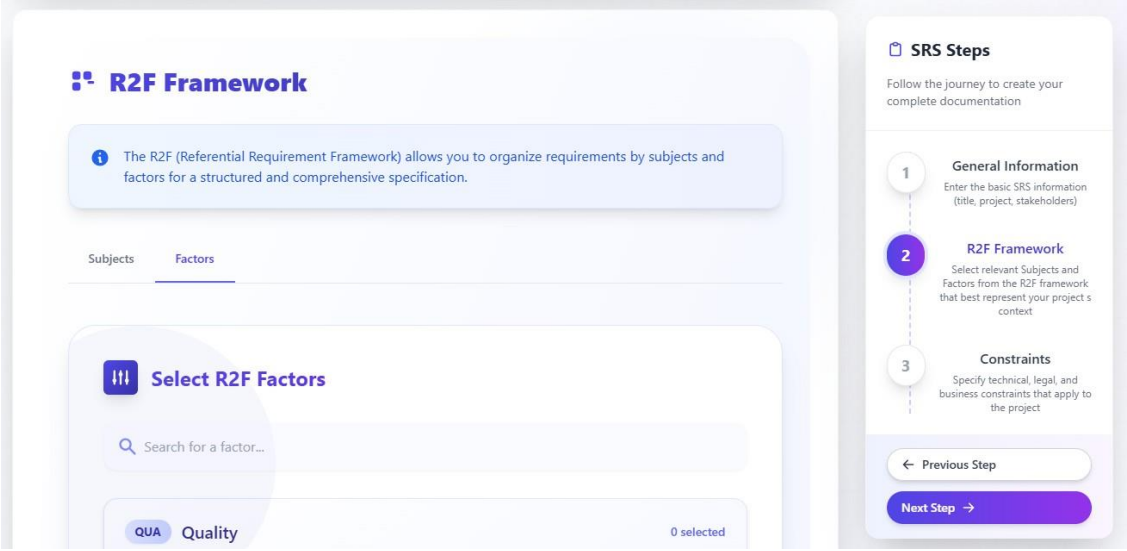



Figure 6.6: Step2: Generic Factors Selection

## Step 3 – Constraints Definition

In the final step (Figure 6.7), the expert defines a set of project basic constraints that will be embedded into the SRS template. Constraints defined at this stage may include technical requirements (e.g., performance, compatibility), legal obligations (e.g., GDPR compliance), or business policies (e.g., budget or delivery limits). The expert uses a popup form—accessed via the "Add Constraint" button—to specify the constraint's rule name, type, description, priority, and status.

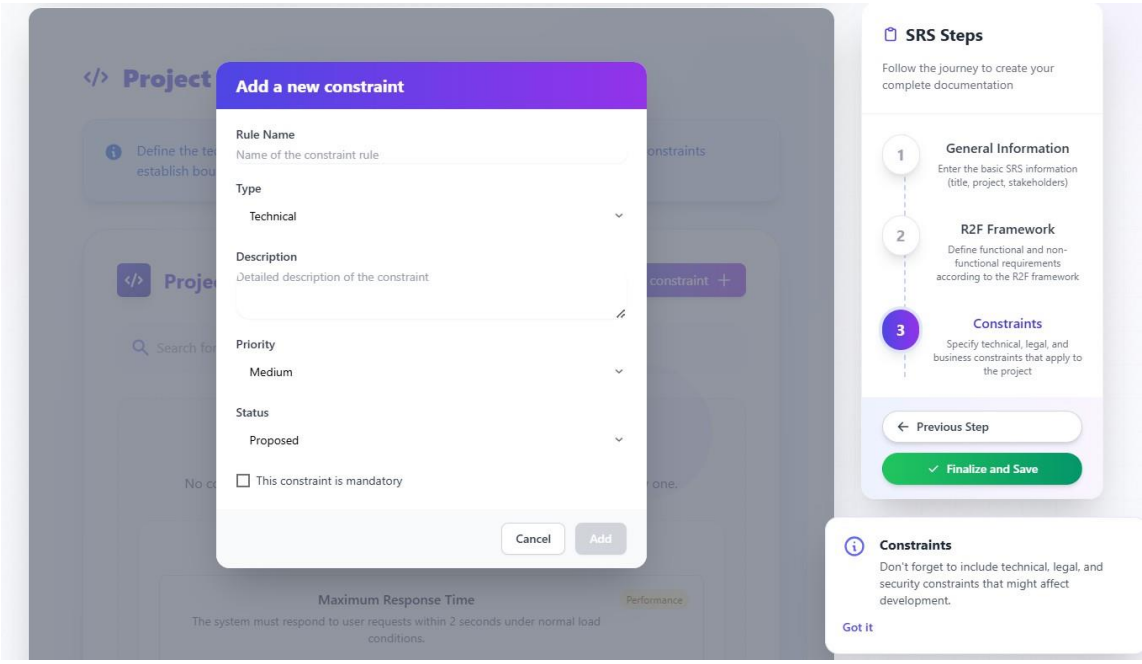



Figure 6.7: Step 3 – Interface for defining constraints

### VI.3.4 SRS Library Interface

The SRS Library (Figure 6.8) interface allows experts to browse, create, edit, and manage Software Requirements Specification (SRS) templates and profiles in one centralized view.
It features summary cards for quick insights, along with filters and search tools for efficient navigation. Each SRS is displayed as a card with key metadata and action buttons for viewing, editing, or deleting.

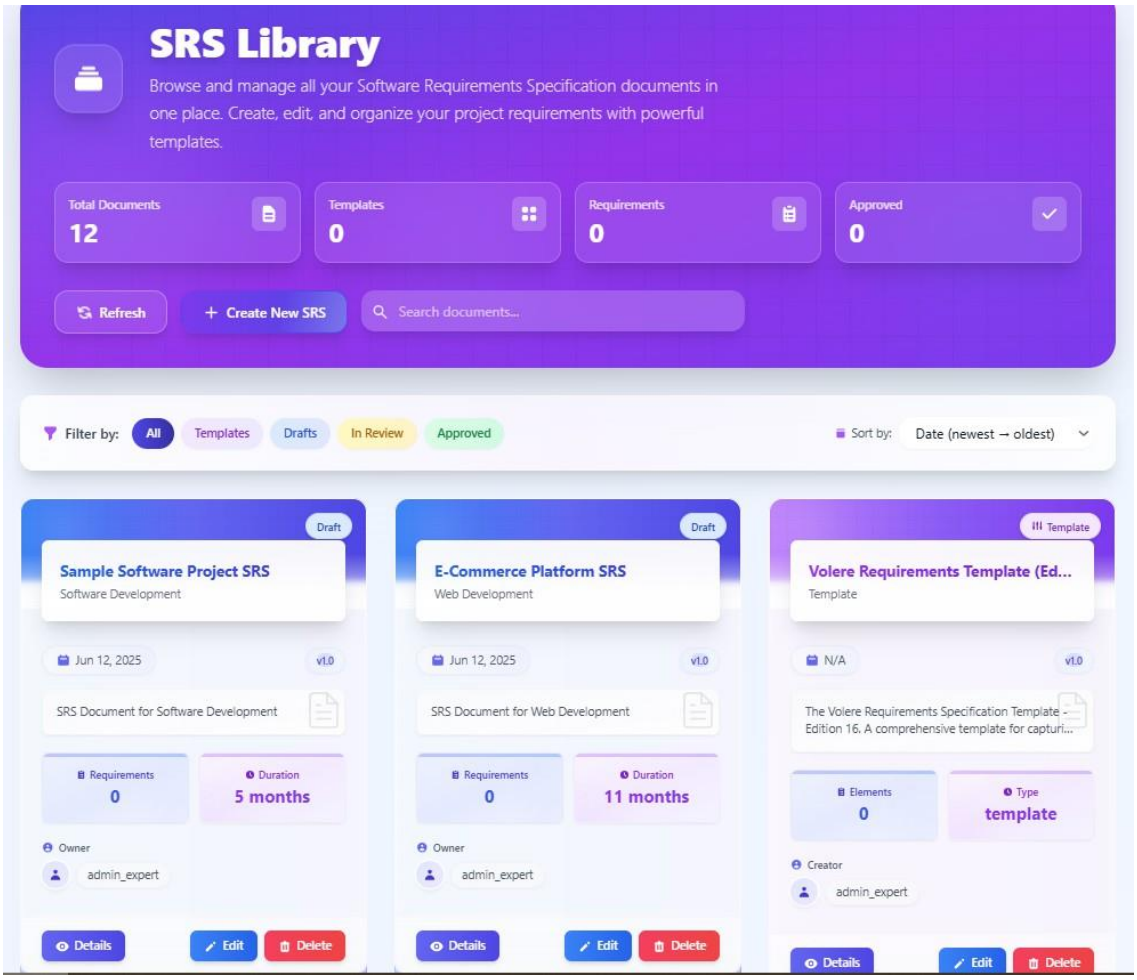



Figure 6.8: SRS Library interface for managing templates and requirement profiles

## VI.3.5 Data Explorer Interface Overview

The Data Explorer interface in ExpertDASH serves as a central hub for navigating and managing all data entities.

The top section in (Figure 6.9) provides summary cards for key entities (Scales, Constraints, Experts, etc.) and a visual Entity Relationship Model. This gives users a quick understanding of system structure and data load status.

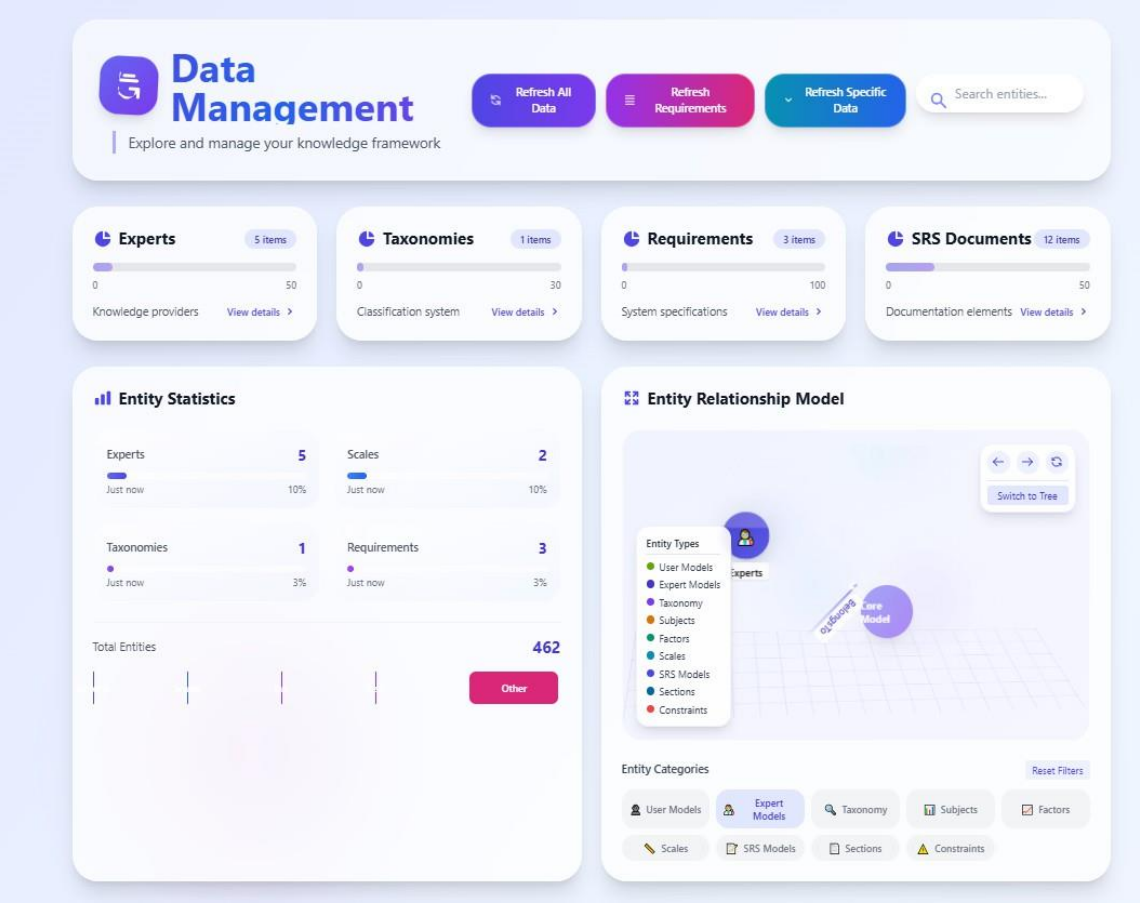



Figure 6.9: Initial interface showing entity overview cards and relationship diagram

Below the overview (Figure 6.10), the expert can filter data by entity type and choose how to visualize it—such as cards, tables, or graphs. This example displays the “Scales” entity in table format, with options to edit or delete each entry.

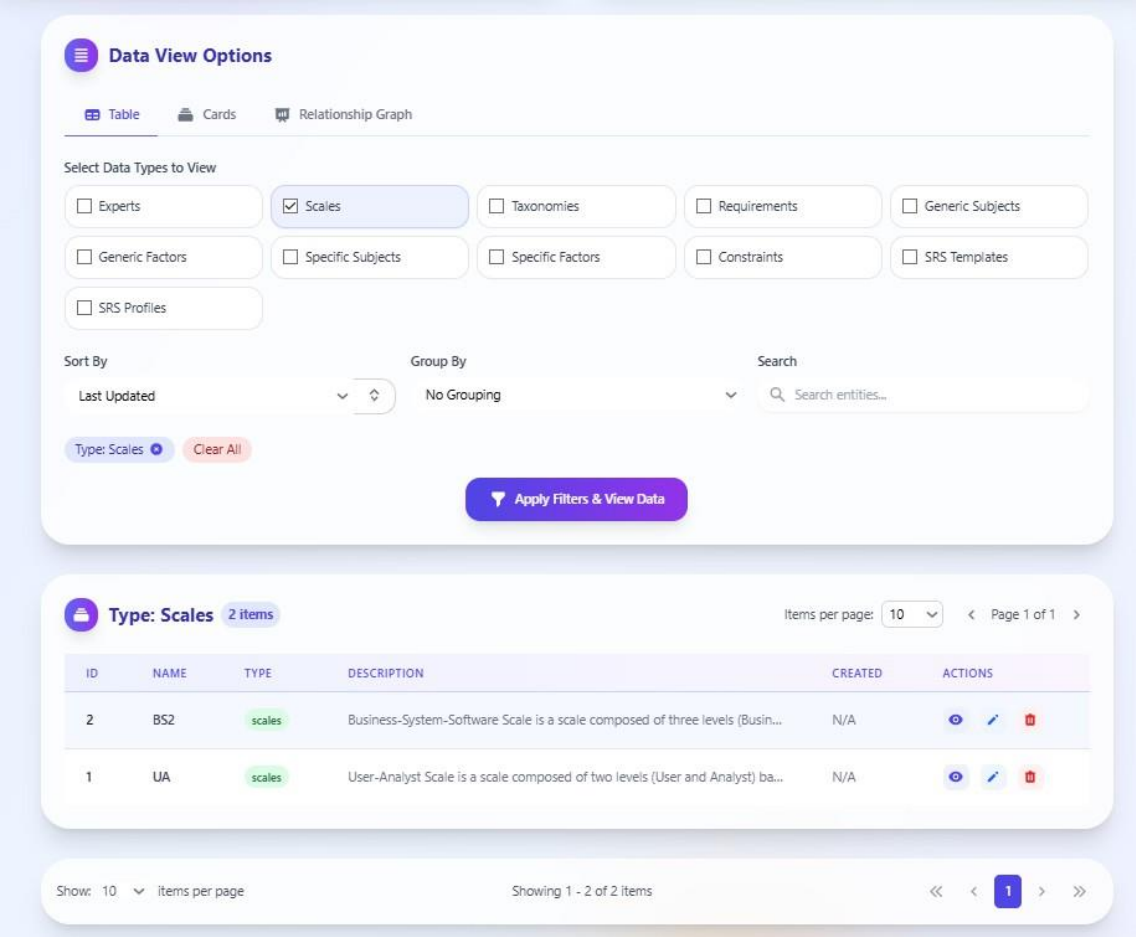



Figure 6.10: Table view showing filtered data for “Scales”

Upon selecting a record (Figure 6.11), all its properties and relationships are displayed in full detail. This supports comprehensive analysis and direct editing or deletion of entity data.

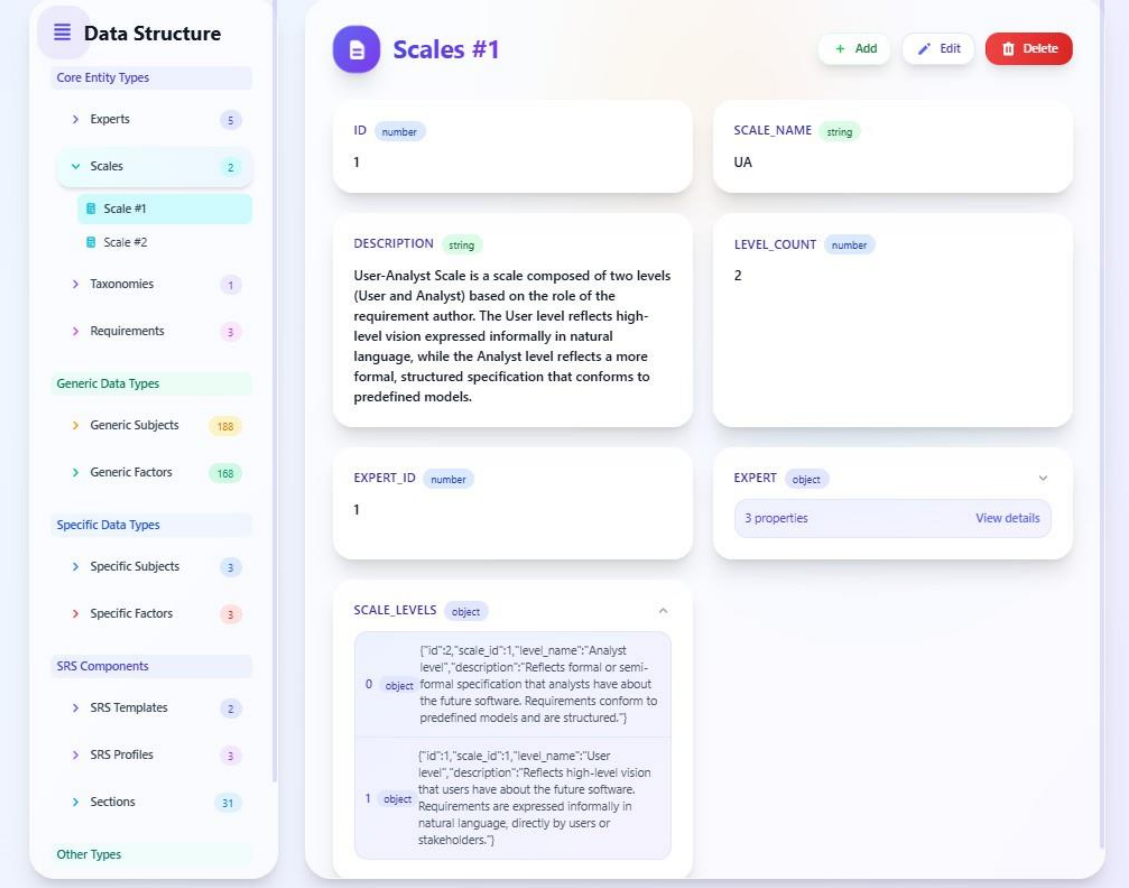



Figure 6.11: Expanded view of a single scale with full properties

### VI.3.6 Settings Interface

The Settings interface in ExpertDASH centralizes user preferences, documentation access, and account management.

This section (Figure 6.12) allows users to update their personal profile details.

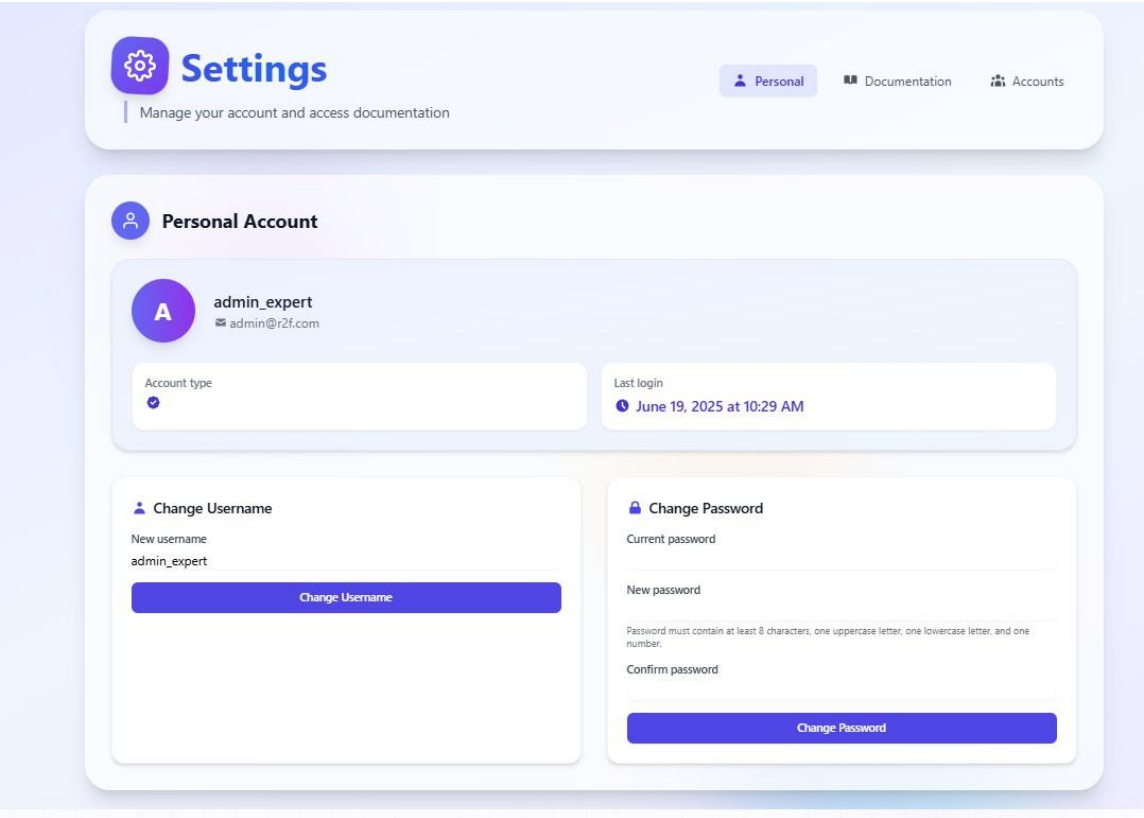



Figure 6.12: Settings – Personal Information Update

This section (Figure 6.13) provides downloadable documentation, including a user guide, technical manual, and frequently asked questions.

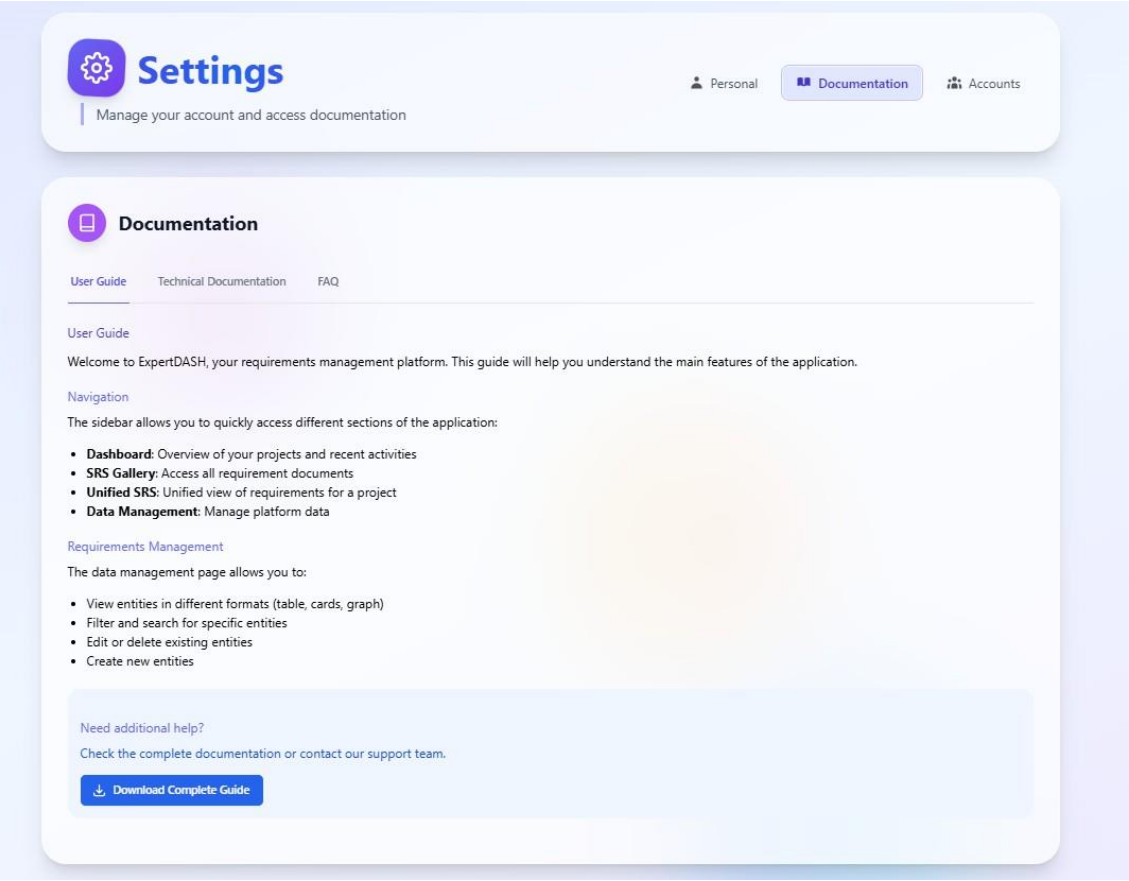



Figure 6.13: Settings – Access to Documentation

Here (Figure 6.14), experts can create new user accounts by entering credentials and optionally assigning expert privileges via a checkbox before confirming the action.

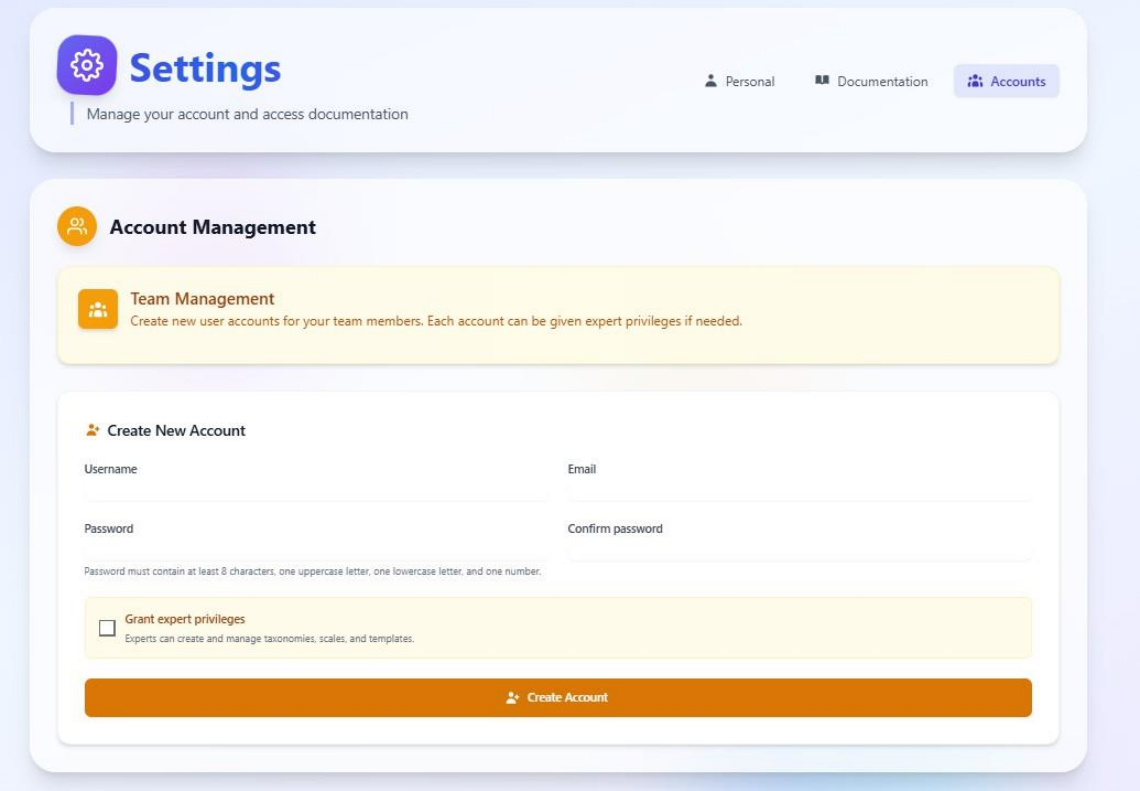



Figure 6.14: Settings – Account Creation for New Users

## VI.4 Conclusion

The implementation of the ExpertDASH system translates the theoretical SMART method into a functional web-based tool. Each interface was carefully developed to support experts in structuring, managing, and refining software requirements. Core modules such as SRS creation, data exploration, and constraint management were realized using modern design principles.

Although the SMART method proposes a hybrid categorization framework, this aspect has not yet been implemented in the current version of the Expert Dashboard. Its integration is deferred to the future Analyst Dashboard due to the current focus on expert-centric functionalities and the added complexity of modeling hybrid categorization logic within the project's development timeframe.

Overall, this implementation lays the groundwork for a complete SMART-based requirements engineering suite.

# VII / Conclusion

This thesis presented the design and implementation of the **Expert Dashboard**, the first functional module of a software tool intended to support the SMART method (Systemic Approach of caTegorizing and Modeling Requirements). Focusing specifically on operationalizing the R2F Framework and creation of SRS while presenting the hybrid categorization model as a theoretical foundation for future extensions.

By translating theoretical models into a practical web-based platform, this work has provided a foundational tool for bridging the gap between informal stakeholder needs and formal specifications. Through features such as SRS creation, constraint management, and structured requirement modeling, the Expert Dashboard demonstrates how methodological rigor can be integrated into real-world requirement engineering practices.

Although the system is currently limited to expert-level functionalities, it establishes a robust technical and conceptual groundwork for future enhancements.

Notably, one promising direction is the development of a complementary **Analyst Dashboard**, which would enable analysts to input and manage structured requirements. This module would complete the SMART method's refinement chain by allowing analysts to classify and trace these requirements using the semantic foundation defined by experts, thus enhancing consistency and reusability in iterative development environments.

Looking beyond immediate extensions, this work also suggests that structured methods like SMART—when backed by supportive tooling—can significantly improve requirement traceability, consistency, and reuse across projects. As the complexity of software systems continues to grow, scalable and formalized approaches to requirement modeling will be increasingly vital.

In conclusion, while this thesis addresses only a subset of the broader SMART vision, it contributes meaningfully to the field of Requirements Engineering by making a theoretical method operational, usable, and extensible. It serves as both a practical artifact and a conceptual bridge between academic innovation and real-world application.

# Bibliography


[1] *Axios - Promise based HTTP client for the browser and node.js*. Accessed on 2025-04-16. 2025. URL: https://axios-http.com/.

[2] Beck et al. *Manifesto for Agile Software Development*. https://agilemanifesto.org. 2001.

[3] M. Borg et al. *Early Requirements Traceability with Domain-Specific Taxonomies: A Pilot Experiment*. Tech. rep. Accessed: 2025-05-19. Swedish Transport Administration (Trafikverket), 2020. URL: https://trafikverket.diva-portal.org/smash/record.jsf?pid=diva2%3A1734383.

[4] M. Borg et al. "Early Requirements Traceability with Domain-Specific Taxonomies". In: *arXiv preprint arXiv:2311.12146* (2023). Accessed: 2025-05-19. URL: https://arxiv.org/pdf/2311.12146.pdf.

[5] A. Chikh. "Requirements Metadata Language (RML): Toward Semantic Categorization of Requirements". In: *International Journal of Computer Science Issues* 14.1 (2017), pp. 34–41.

[6] A. Chikh. *Systemic Approach to Categorizing and Modeling Requirements*. Wiley-ISTE, 2025. ISBN: 978-1-394-37273-7. URL: https://www.wiley.com/en-us/Systemic%2BApproach%2BCategorizing%2Band%2BModeling%2BRequirements-p-9781394372737.

[7] Lawrence Chung et al. *Non-Functional Requirements in Software Engineering*. Springer, 2012.

[8] Jane Cleland-Huang, Rossana Settimi, and Yijun Zou. "Software Traceability: How the Past Can Help the Future". In: *IEEE Software* 24.4 (2007), pp. 40–47. DOI: 10.1109/MS.2007.104.

[9] Jane Cleland-Huang et al. "Software traceability: Trends and future directions". In: *Proceedings of the Future of Software Engineering* (2012), pp. 55–69.

[10] *GitHub: Where the world builds software*. Accessed on 2025-04-20. 2025. URL: https://github.com/.

[11] Olly Gotel and Anthony Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering* (1994), pp. 94–101.

[12] *IEEE Std 830-1998 - IEEE Recommended Practice for Software Requirements Specifications*. Consulted on 17/05/2025. IEEE Computer Society, 1998. URL: https://standards.ieee.org/standard/830-1998.html.

[13] *ISO/IEC 25010:2011 - Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*. https://iso.org/standard/35733.html. ISO/IEC, 2011.

[14] *ISO/IEC/IEEE 29148:2018 – Systems and software engineering – Life cycle-processes-Requiremen engineering*. Consulted on 17/04/2025. URL: https://www.iso.org/standard/72089.html..

[15] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, 1998.

[16] *Laravel - The PHP Framework For Web Artisans*. Accessed on 2025-04-20. 2025. URL: https://laravel.com/.

[17] Soren Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, 2002.

[18] *MySQL :: The World's Most Popular Open Source Database*. Accessed on 2025-04-20. 2025. URL: https://www.mysql.com/.

[19] Bashar Nuseibeh and Steve Easterbrook. "Requirements Engineering: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2000, pp. 35–46.

[20] *PHP: Hypertext Preprocessor*. Accessed on 2025-04-20. 2025. URL: https://www.php.net/.

[21] *phpMyAdmin - About*. Accessed on 2025-04-20. 2025. URL: https://www.phpmyadmin.net/about/.

[22] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010.

[23] *Postman | API Development Environment*. Accessed on 2025-04-20. 2025. URL: https://www.postman.com/.

[24] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-Wesley, 2012.

[25] Cezar Sas and Andrea Capiluppi. "Automatic Bottom-Up Taxonomy Construction: A Software Application Domain Study". In: *arXiv preprint arXiv:2409.15881* (2024). Accessed: 2025-05-19. URL: https://arxiv.org/html/2409.15881v1.

[26] Ian Sommerville. *Software Engineering*. 10th ed. Pearson, 2016.

[27] *Tailwind CSS — A Utility-First CSS Framework*. Accessed on 2025-04-20. 2025. URL: https://tailwindcss.com/.

[28] *Types of Stakeholders and Their Roles: A Quick Reference Guide*. Consulted on 17/05/2025. 2025. URL: https://simplystakeholders.com/types-of-stakeholders-and-their-roles/.

[29] Michael Unterkalmsteiner. "Early Requirements Traceability with Domain-Specific Taxonomies - A Pilot Experiment". In: *Proceedings of the 28th IEEE International Requirements Engineering Conference (RE)*. Zurich, Switzerland: IEEE, 2020, pp. 322–327.

[30] *Visual Studio Code*. Accessed on 2025-04-20. 2025. URL: https://code.visualstudio.com/.

[31] *Vue.js — The Progressive JavaScript Framework*. Accessed on 2025-04-20. 2025. URL: https://vuejs.org/.

[32] *WAMPServer*. Accessed on 2025-04-20. 2025. URL: https://www.wampserver.com/en/.

[33] Karl Wiegers and Joy Beatty. *Software Requirements*. 3rd. Microsoft Press, 2013.

# Abstract

**Abstract**
This thesis focuses on the development and evaluation of the first functional module of a system supporting the SMART method (Systemic Approach of caTegorizing and Modeling Requirements), called *ExpertDASH*. This expert-centered web application is designed to facilitate the structuring of early-stage requirements through hierarchical models such as the R2F Framework (Generic Subjects and Influencing Factors). The ExpertDASH platform enables experts to define the semantic foundation that will later be used in additional modules such as *AnalystDASH*, where analysts will complete the requirement lifecycle. The system was implemented using Laravel, Vue.js, and MySQL, following an MVC architecture for scalability and modularity.

**Keywords:** SMART method, Requirements Engineering, R2F Framework, SRS, Laravel, Vue.js, Expert Dashboard.

**Résumé**
Ce mémoire porte sur le développement et l'évaluation du premier module fonctionnel d'un système de support à la méthode SMART (Approche Systémique de Classification et Modélisation des Exigences), nommé *ExpertDASH*. Cette application web, dédiée aux experts, facilite la structuration des exigences à travers des modèles hiérarchiques tels que le cadre R2F (Sujets Généraux et Facteurs). La plateforme permet à l'expert de définir la base conceptuelle qui sera exploitée ultérieurement dans des modules comme *AnalystDASH*, destiné aux analystes pour l'expression complète des exigences. Le système a été développé avec Laravel, Vue.js et MySQL, selon une architecture MVC assurant la maintenabilité et l'évolutivité.

**Mots-clés :** Méthode SMART, Ingénierie des exigences, Cadre R2F, SRS, Laravel, Vue.js, Tableau de bord expert.

**ملخص**
يركز هذا العمل على تصميم وتطوير أول وحدة من نظام يدعم منهجية لتصنيف ونمذجة المتطلبات، تحت اسم نظام موجه للخبراء في هندسة المتطلبات. تهدف هذه الوحدة إلى تنظيم المتطلبات الأولية باستخدام نماذج هرمية مثل إطار الموضوعات العامة والعوامل المؤثرة والتصنيف الهجين (حسب المصدر أو التجريد أو الطبيعة). يساعد التطبيق الخبراء على تحديد هيكل مفاهيمي يمكن استخدامه لاحقًا في وحدات أخرى، التي ستمكن المحللين من كتابة متطلبات قابلة للتنفيذ. تم بناء النظام باستخدام أدوات وأطر حديثة، ويعتمد على نمط هيكلي يسهل التعديل والتوسعة

**الكلمات المفتاحية** : هندسة المتطلبات، تصنيف، نمذجة، تطبيق ويب، نظام دعم ، ***SMART***