الجمهورية الشعبية الديمقراطية الجزائرية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
المدرسة العليا للإعلام الآلي 08 ماي 1945. بسيدي بلعباس
École Supérieure en Informatique
-08 Mai 1945- Sidi Bel Abbès

THESIS

To obtain the diploma of **Engineer**
Field : **Computer Science**
Specialty : **Systèmes d'Information et Web (SIW)**

Theme

Computer Science Book Validation : A Structured Approach

Presented by :
YASSER Melki

Graduation Date : **Oct, 2024**
In front of the jury composed of :

Dr ELOUALI Nadia	President
Pr. Stéphane Ducasse	Supervisor
Dr. Awad Samir	Supervisor
Dr KLOUCHE Badia	Examiner

Academic Year : 2023/2024

Dédicace

À mon père,
Ta sagesse, ta force et ton soutien indéfectible ont été les fondations sur lesquelles j'ai construit tout ce que je suis. Tu étais une lumière guidante, toujours avec des mots d'encouragement, et tu m'as appris la valeur de la persévérance, de l'humilité et de la bienveillance. Bien que tu ne sois plus parmi nous, ta présence continue de remplir mon cœur et d'influencer chacune de mes décisions. Merci d'avoir été un père extraordinaire, et je porterai ton héritage avec moi pour toujours.

À ma mère,
Ton amour infini, tes soins et tes sacrifices ont été les piliers de mon parcours. Tu as toujours cru en moi, et ta force continue de m'inspirer chaque jour. Ta douce guidance et ton affection ont fait de moi la personne que je suis aujourd'hui, et je te serai éternellement reconnaissant pour les précieuses leçons que tu m'as enseignées. Merci d'être encore cette force constante qui me garde les pieds sur terre.

À mes merveilleux frères et sœurs,
Vous avez été mes compagnons à chaque étape de ce voyage. Votre soutien constant, vos encouragements et votre foi en mes capacités m'ont donné la force de surmonter chaque défi. Vous avez toujours été là, prêts à m'aider quand j'en avais besoin, et pour cela, je vous en suis infiniment reconnaissant. Je suis chanceux de vous avoir dans ma vie, et cette réussite vous appartient autant qu'à moi.

À vous tous,
Ce travail vous est dédié de tout cœur, en signe de l'immense gratitude et de

l'amour que je ressens pour chacun de vous.

Remerciement

Avec une profonde appréciation et humilité, je reconnais les bénédictions infinies que m'a accordées Allah (SWT), le Tout Miséricordieux et le Très Miséricordieux. C'est par Sa grâce divine que j'ai reçu la force, le savoir et la persévérance nécessaires pour mener à bien cette étude.

Je tiens à exprimer ma sincère gratitude à mon directeur de thèse, le Professeur Awad Samir à l'ESI SBA, pour son encadrement et son soutien tout au long de ce parcours.

Je souhaite également exprimer ma profonde reconnaissance au Dr. Stéphane Ducasse, Directeur de Recherche au Centre Inria de l'Université de Lille. Son aide précieuse et son orientation ont été essentielles à mon parcours et à la réussite de ce projet. L'expertise, la bienveillance et le mentorat du Dr. Ducasse ont enrichi mon expérience de manière inestimable, m'offrant des perspectives et des connaissances que je n'aurais jamais pu acquérir seul. Sa capacité à combiner compétence académique et soutien sincère, tant dans nos interactions professionnelles que personnelles, a été véritablement remarquable. Je lui suis profondément reconnaissant pour ses contributions et pour les compétences qu'il m'a aidé à développer. Mes remerciements vont également à tous les membres de l'équipe Evref pour leur soutien constant et leur collaboration.

Je suis profondément honoré que les membres éminents du jury aient pris le temps de réviser mon travail. Votre dévouement est sincèrement apprécié.

Ma plus profonde gratitude s'étend au corps enseignant et administratif dévoué de l'ESI, qui nous ont offert une excellente éducation.

Enfin, j'adresse mes remerciements les plus chaleureux à toutes les personnes qui, de quelque manière que ce soit, directement ou indirectement, ont contribué à l'achèvement de ce travail.

Abstract

The report focuses on addressing structural and semantic challenges in validating large-scale technical documentation written in Microdown, a lightweight markup language used within the Pharo ecosystem. The thesis presents the design and implementation of an automated Book Validation Tool, aimed at ensuring the integrity and correctness of documents. The tool systematically checks for structural problems like missing files, cyclic references, and anchor issues, as well as semantic problems such as code block errors and inconsistencies. The solution utilizes advanced validation strategies, applying design patterns such as Strategy and Visitor to enhance flexibility and scalability. Through continuous testing and validation, the tool supports documentation consistency, particularly in evolving codebases.

Keywords— Documentation validation, Microdown, structural validation, semantic validation, Pharo, design patterns, automated tool

الملخص

يركز هذا التقرير على معالجة التحديات الهيكلية والدلالية في التحقق من صحة الوثائق التقنية واسعة النطاق المكتوبة بلغة الترميز الخفيفة ، وهي لغة تُستخدم ضمن نظام البيئي. يقدم البحث تصميم وتنفيذ أداة التحقق الآلي من صحة الكتب، بهدف ضمان سلامة وصحة المستندات. تقوم الأداة بالتحقق المنهجي من المشكلات الهيكلية مثل الملفات المفقودة، المراجع الدورية، ومشاكل الروابط، وكذلك المشكلات الدلالية مثل أخطاء التعليمات البرمجية وعدم الاتساق. يستخدم الحل استراتيجيات تحقق متقدمة، مطبقًا أنماط تصميم مثل "الاستراتيجية" و"الزائر" لتعزيز المرونة وقابلية التوسع. من خلال الاختبار والتحقق المستمر، تدعم الأداة اتساق الوثائق، خاصةً في قواعد التعليمات البرمجية المتطورة

الكلمات المفتاحية: لتحقق من صحة الوثائق، التحقق الهيكلي، التحقق الدلالي، أنماط التصميم، الأداة الآلية.

Résumé

Le rapport se concentre sur la résolution des défis structurels et sémantiques dans la validation de la documentation technique à grande échelle écrite en Microdown, un langage de balisage léger utilisé dans l'écosystème Pharo. Le mémoire présente la conception et la mise en œuvre d'un outil de validation automatisée des livres, visant à garantir l'intégrité et la précision des documents. L'outil vérifie systématiquement les problèmes structurels tels que les fichiers manquants, les références cycliques et les problèmes d'ancrage, ainsi que les problèmes sémantiques tels que les erreurs dans les blocs de code et les incohérences. La solution utilise des stratégies de validation avancées en appliquant des motifs de conception tels que la Stratégie et le Visiteur pour améliorer la flexibilité et l'évolutivité. Grâce à des tests et des validations continus, l'outil soutient la cohérence de la documentation, notamment dans les bases de code en évolution. ***Keywords***— Validation de la documentation, Microdown, validation structurelle, validation sémantique, Pharo, motifs de conception, outil automatisé.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

CHAPTER 1

INTRODUCTION

1.1 Context

The modern software development lifecycle necessitates an increasingly automated approach to ensure that documentation is kept up to date and consistent with the rapidly evolving codebase. In particular, in agile environments, where iterative development and frequent updates are the norm, maintaining accurate and reliable documentation becomes a critical task. In this context, the Microdown framework emerges as a robust solution for generating and validating documents written in a lightweight markup language. Microdown is particularly well-suited for producing technical documentation due to its simplicity, ease of integration, and compatibility . However, as with any document generation tool, ensuring that the resulting output is valid, complete, and correctly structured is crucial for its long-term viability in professional settings. Therefore, building an automated validation tool within the Microdown ecosystem is essential to verify that documentation meets specified standards and requirements.

1.2 Problem Statement

Despite the advantages that Microdown offers, a significant challenge in document-driven development environments is ensuring the validity and correctness of documents over time. As software projects scale, the risk of documentation becoming outdated, inaccurate, or inconsistent with the evolving codebase increases. This leads to potential errors in the documentation, miscommunication among teams, and a reduction in overall project quality. Without an automated mechanism to

validate documents, manual checks become prone to oversight, inefficiency, and human error.

1.3 Objective Of The Thesis

The objective of this thesis is to develop a Book Validation Tool integrated into the Microdown system. The tool is designed to ensure that documents, particularly books or extended technical reports written in Microdown, adhere to predefined validation rules. By automating the verification process, it systematically checks for issues such as incorrect syntax, incomplete sections, and adherence to structural guidelines. This ensures that each document remains consistent and accurate, even as the codebase evolves. The tool also provides continuous validation throughout the software lifecycle, promoting reliability and precision in documentation

Part II

Background

CHAPTER 2
BACKGROUND

2.1 Introduction

In today's fast-paced technological landscape, effective communication is paramount for the success of any software project. As software systems grow in complexity and scale, the need for clear and accurate documentation becomes increasingly essential [11]. Documentation serves as a vital component of the software development lifecycle, providing a reference point for developers, users, and stakeholders alike. It not only aids in understanding and navigating the intricacies of code but also ensures that everyone involved in a project is aligned on its goals, functionality, and requirements.

In computer science, documentation encompasses a variety of forms, including user manuals, technical specifications, API documentation, and inline comments within code. Each type of documentation plays a distinct role in facilitating knowledge transfer, enhancing user experience, and supporting ongoing maintenance efforts. However, the rapid evolution of technology, coupled with agile development practices, poses significant challenges to maintaining high-quality documentation [20]. As a result, validating the accuracy, completeness, and relevance of documentation has become a critical focus area for software teams.

This report explores the structured approach to validating computer science documentation, emphasizing the importance of thorough validation processes. By understanding the role of documentation in software development and the challenges associated with its validation, we can implement strategies that enhance the overall quality of technical documentation, ultimately leading to more successful software projects.

2.2 Importance of Documentation in Computer Science

Documentation plays a critical role in the field of computer science, serving as a foundational element that bridges the gap between complex systems and their users [6]. It provides essential insights into software architecture, design choices, and implementation details, making it easier for developers, users, and stakeholders to understand and interact with the software. In software development, accurate documentation acts as a roadmap, guiding developers through the intricacies of the codebase. This is particularly vital in large projects or team settings, where multiple contributors are involved. Good documentation fosters collaboration, reduces onboarding time for new team members, and enhances overall productivity by enabling developers to find relevant information quickly.

Moreover, documentation is indispensable for end users who rely on clear instructions to effectively use software products. Well-crafted user manuals and API documentation ensure that users can leverage software features without encountering confusion or frustration. In regulated industries, thorough documentation is not just a best practice but a legal requirement, as it aids compliance with standards and protocols [10]. Additionally, comprehensive documentation plays a significant role in software maintenance, as it allows developers to make informed decisions when updating or modifying existing code. Without proper documentation, the knowledge of how a system works may become fragmented, leading to difficulties in troubleshooting and prolonging development cycles. Overall, the importance of documentation in computer science cannot be overstated; it is essential for promoting clarity, facilitating communication, and ensuring the long-term success of software projects.

2.3 Challenges in Documentation Validation

Validating documentation in the field of computer science presents several challenges that can hinder effective quality assurance. Key challenges include:

- **Dynamic Nature of Software Development:** In agile environments, frequent iterations and updates complicate the alignment of documentation with the evolving codebase, often resulting in discrepancies between the documentation and the actual functionality of the software [2].
- **Complexity of Technical Documentation:** Different types of documentation, such as user manuals, API references, and design documents, possess unique structures and conventions, making it challenging to ensure that each

type adheres to specific validation criteria, especially when considering the diverse audiences they serve, including developers, end users, and stakeholders [14].

- **Diversity of Documentation Formats:** The existence of documentation in various formats—such as text files, HTML pages, wikis, and interactive help systems—complicates the validation process, as each format requires distinct approaches and tools, increasing the complexity of ensuring consistency and quality across all documentation.
- **Subjectivity in Validation Criteria:** Validation criteria for what constitutes "complete" or "accurate" documentation can be subjective and vary significantly across projects and stakeholders, leading to conflicting opinions on documentation quality and potentially resulting in disagreements that hinder effective validation efforts [19].
- **Integration of Automated Validation Tools:** Incorporating automated validation tools into existing workflows necessitates careful consideration to prevent disruptions in the development process, and ensuring that these tools are effectively utilized and accepted by team members can pose significant challenges.
- **Resource Constraints:** Teams often face limitations in time and personnel, making it challenging to prioritize thorough documentation validation; balancing the need for documentation quality with other project demands can lead to compromises in the validation process.
- **Evolving Standards and Best Practices:** As documentation standards and best practices continuously evolve, teams must remain updated on the latest guidelines, and adapting to new standards can create challenges in ensuring that existing documentation meets current expectations [17].

2.4 Key Validation Criteria

To effectively validate technical documentation, several key criteria must be established to ensure quality and reliability.

- **Syntactic Correctness:** This first criterion focuses on grammar, punctuation, and formatting rules. Documentation should be free from typographical errors and adhere to predefined formatting styles. This ensures that the documentation is not only readable but also professional in appearance, fostering trust among users [9].

- **Completeness:** This is another critical validation criterion. Documentation should include all necessary information relevant to its purpose, ensuring that users can find the details they need without ambiguity. This involves verifying that each section of the documentation is present and sufficiently detailed. For instance, API documentation must include descriptions of endpoints, parameters, return values, and example requests [15]. Incomplete documentation can lead to confusion and misuse of the software, ultimately diminishing user satisfaction.

- **Relevance and Accuracy:** These are equally important. Documentation should accurately reflect the current state of the codebase and its functionalities. This necessitates regular updates to documentation as the software evolves, ensuring that users are provided with the most current and applicable information. Moreover, validation should also consider whether the content is relevant to the target audience; for example, documentation aimed at developers may differ significantly from that intended for end users [18].

- **Consistency:** Consistency across the documentation is crucial for maintaining a coherent user experience. This includes using consistent terminology, formatting, and style throughout the documentation. Inconsistencies can lead to confusion and misinterpretation, undermining the documentation's effectiveness. Establishing a style guide or employing automated tools to enforce consistency can help address this issue [13]. By adhering to these key validation criteria—syntactic correctness, completeness, relevance and accuracy, and consistency—organizations can significantly improve the quality and reliability of their technical documentation, thereby enhancing the overall user experience and facilitating better software development practices.

- **Overall Documentation Quality:** The interplay of these criteria ultimately contributes to the overall quality of technical documentation, which is vital for successful software development. High-quality documentation fosters effective communication, aids user comprehension, and supports ongoing maintenance efforts [4].

2.5 Design Patterns for Validation Implementation

2.5.1 The Strategy Design Pattern

The Strategy Design Pattern is one of the behavioral design patterns that allows algorithms to be selected at runtime [5]. This pattern defines a family of algorithms,

encapsulates each one as an individual class, and makes them interchangeable. The key principle behind this pattern is that it separates the algorithm's behavior from the client that uses it, enabling different behaviors without altering the client.

In the context of software development, this design pattern is particularly useful when multiple approaches or strategies can be employed to solve a particular problem. For instance, in a book validation system, different validation algorithms can be applied based on various factors such as format, metadata consistency, or content structure. By applying the Strategy Design Pattern, the validation logic can be easily swapped, extended, or modified without affecting the core validation flow.

2.5.1.1 Components of the Strategy Design Pattern

The Strategy Design Pattern consists of three key components:

- **Context:** The context is the class that uses the strategy. It contains a reference to one of the strategy classes and delegates the behavior to it. In your case, this could be a `BookValidator` class responsible for validating books.
- **Strategy Interface:** This is an interface that defines the behavior or the strategy that can be implemented by various classes. In the book validation example, a `ValidationStrategy` interface could define a method like `validate(Book)`.
- **Concrete Strategies:** These are classes that implement the `Strategy` interface. Each class represents a different way of validating a book, such as `FormatValidation`, `MetadataValidation`, or `ContentValidation`.

2.5.1.2 Applying the Strategy Pattern in Book Validation

In Pharo's book validation system, the Strategy Pattern provides a flexible way to apply different validation strategies depending on the type of book or the specific validation required. For instance, one book might require metadata consistency checks, while another might require semantic or evolution validation to ensure that the code example are consistent.

By using this pattern, we can easily add new validation strategies without modifying the main `BookValidator` class. This makes the system more scalable and easier to maintain. Below is a conceptual representation of how different strategies can be employed in the book validation system:

```
public interface ValidationStrategy {
```

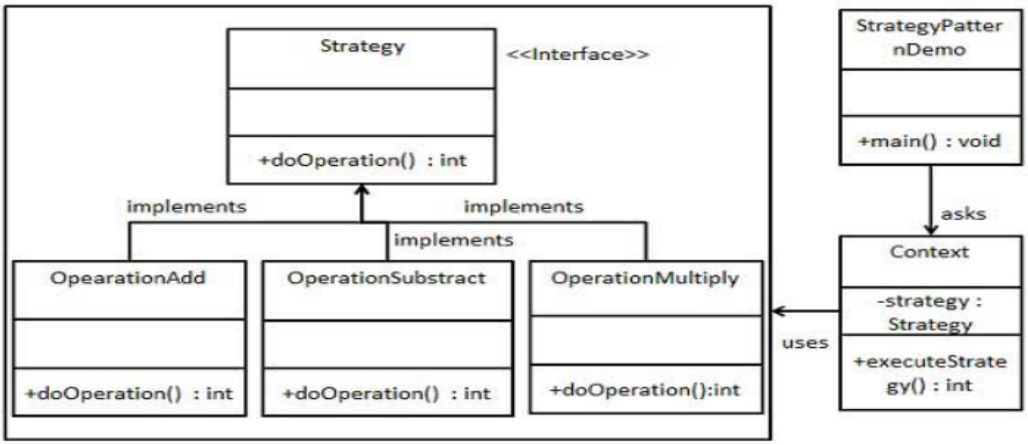

Figure 2.1: Strategy Design Pattern

```
    boolean validate(Book book);
}

public class FormatValidation implements ValidationStrategy {
    public boolean validate(Book book) {
        // Check the format of the book
        return isValidFormat(book);
    }
}

public class MetadataValidation implements ValidationStrategy {
    public boolean validate(Book book) {
        // Validate book metadata, such as title and author
        return isValidMetadata(book);
    }
}

public class ContentValidation implements ValidationStrategy {
    public boolean validate(Book book) {
        // Validate the content structure of the book
        return isValidContent(book);
    }
}

public class BookValidator {
```

```
    private ValidationStrategy strategy;

    public BookValidator(ValidationStrategy strategy) {
        this.strategy = strategy;
    }

    public boolean validate(Book book) {
        return strategy.validate(book);
    }
}
```

2.5.1.3 Advantages of the Strategy Pattern in Book Validation

Using the Strategy Pattern in the book validation system provides several advantages:

- **Open/Closed Principle:** The system adheres to the open/closed principle because new validation strategies can be added without modifying the existing validation logic.
- **Reusability:** The strategy classes can be reused across different contexts, not just limited to book validation. This promotes modularity and code reuse.
- **Simplified Code Management:** Since each validation strategy is encapsulated in its own class, the code becomes easier to manage and debug.
- **Dynamic Behavior:** The validation strategy can be dynamically changed at runtime, allowing for flexible validation rules depending on the book type or specific user requirements.

2.5.1.4 Example Table of Validation Strategies

The following table 2.1 summarizes different validation strategies that could be applied in your book validation system:

2.5.1.5 Conclusion

The Strategy Design Pattern offers an elegant solution to managing different validation requirements in a book validation system. By decoupling the validation logic into separate strategies, the system becomes highly extensible and easy to maintain. As new validation needs arise, they can be incorporated by simply adding new strategies without altering the core functionality of the system.

Validation Strategy	Description	Use Case
Format Validation	Ensures book format follows the required standard	E-books, PDFs
Metadata Validation	Checks metadata fields like title, author	General book publishing
Content Validation	Validates chapter order, headings, content flow	Academic or technical books

Table 2.1: Different validation strategies in the book validation system

2.5.2 Double Dispatch

Double dispatch is a technique in object-oriented programming that allows a method to be dynamically chosen at runtime based on the runtime types of two objects [8]. This differs from the more common single dispatch, where a method is selected based only on the runtime type of the receiver object. In double dispatch, both the receiver object and the argument passed are used to determine which method to invoke, adding flexibility in how behaviors are handled depending on the combination of objects involved.

Double dispatch becomes particularly useful when there are multiple interacting objects that need to collaborate and decide their behavior based on each other's types. A common use case is in games, simulations, or graphical environments, where different entities interact with one another in different ways depending on their type.

In Pharo, double dispatch can be implemented using a combination of method calls between two objects, where each object takes responsibility for invoking the appropriate behavior based on the type of the other object.

2.5.2.1 Pharo Example of Double Dispatch

Consider an example where we have different shapes and a drawing tool that behaves differently depending on the type of shape. Below is a Pharo implementation of double dispatch:

```
Object subclass: #Shape
    instanceVariableNames: ''.

Shape subclass: #Circle
    instanceVariableNames: ''.
```

```
Shape subclass: #Square
    instanceVariableNames: ''.

Shape subclass: #DrawingTool
    instanceVariableNames: ''.

Shape >> interactWith: aShape
    "This is the generic interaction method for shapes"
    aShape interactWithShape: self.

Circle >> interactWithShape: aShape
    "Double dispatch for Circle interacting with other shapes"
    ^ 'Circle interacts with ', aShape class name.

Square >> interactWithShape: aShape
    "Double dispatch for Square interacting with other shapes"
    ^ 'Square interacts with ', aShape class name.

DrawingTool >> interactWith: aShape
    "Drawing tool interacting with different shapes"
    aShape interactWithShape: self.

"Usage Example"
| circle square drawingTool |
circle := Circle new.
square := Square new.
drawingTool := DrawingTool new.

Transcript show: (circle interactWith: square); cr.
Transcript show: (drawingTool interactWith: circle); cr.
```

In this Pharo example, the interaction is dynamically determined based on the type of the ‘Shape‘ objects. The ‘interactWith:‘ method delegates the decision to the second object (via ‘interactWithShape:‘), thus achieving double dispatch.

2.5.2.2 Common Confusion: Double Dispatch vs Visitor Pattern

Double dispatch is often confused with the Visitor Design Pattern due to their similar mechanisms. However, there are key differences between the two:

- **Double Dispatch**: The main focus is on determining the method to execute

based on the runtime types of two interacting objects. The interaction is tightly coupled between these objects.

- **Visitor Pattern**: The pattern focuses on defining a new operation on a class structure without changing the classes themselves. It is useful when multiple distinct operations are required, but the class structure should remain untouched.

Both techniques allow dynamic method resolution based on types, but **Visitor Pattern** abstracts the operation from the object hierarchy, while **Double Dispatch** keeps the operation embedded within the classes.

2.5.2.3 Illustration of Double Dispatch

Figure 2.2 will be included to visually illustrate the process of double dispatch , showing how the interaction flows between two objects dynamically based on their types.

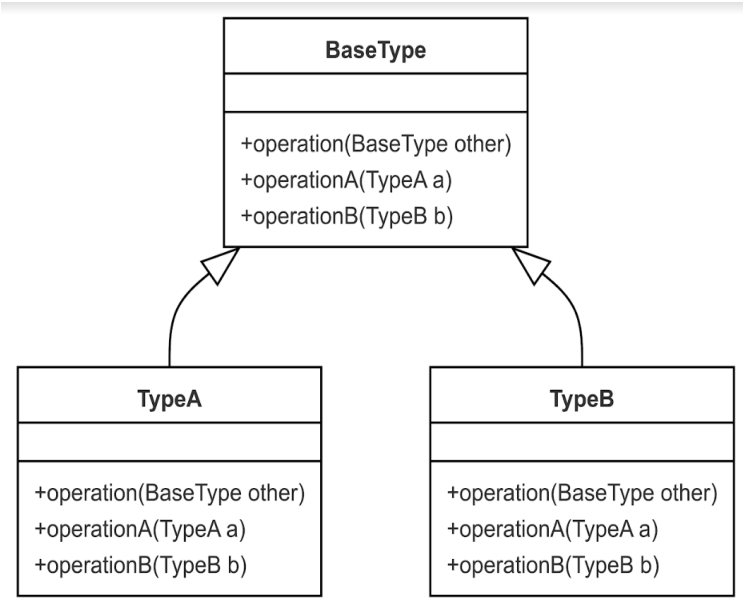

Figure 2.2: Illustration of Double Dispatch

2.5.3 Visitor Design Pattern

The **Visitor Design Pattern** is one of the most powerful and flexible behavioral design patterns. It is particularly useful when you need to perform operations on

a group of objects with varying types [16], and when these operations are expected to evolve without requiring changes to the objects themselves. The Visitor Pattern separates algorithms from the objects on which they operate, making it easier to add new operations without modifying the underlying structure of the objects.

The basic structure of the Visitor Design Pattern consists of:

- **Visitor Interface:** Defines a visit operation for each concrete element class.
- **Concrete Visitor:** Implements the operations to be performed on each concrete element.
- **Element Interface:** Defines the accept method, which takes a visitor object as a parameter.
- **Concrete Element:** Implements the accept method to invoke the appropriate visit method on the visitor.

This pattern is highly effective when you need to perform multiple operations across a class hierarchy without changing the classes themselves.

In my **book validation system**, the Visitor Pattern was indispensable for accessing specific elements (e.g., validating different parts of the book, such as title, author information, and chapters) and applying operations based on the element type. It would have been much more difficult to implement these operations using other methods, as it allowed for seamless extension of functionality by simply adding new visitor operations without altering the existing code base.

2.5.3.1 Key Benefits of the Visitor Pattern

- **Separation of Concerns:** By moving operations into visitor objects, the code in the elements remains focused on data representation, while visitors focus on operations.
- **Extensibility:** New operations can be added easily by defining new visitors without modifying the element classes.
- **Single Responsibility Principle:** The element classes are responsible only for storing data, while the visitor classes handle the operations performed on that data.

2.5.3.2 The Role of Visitor Pattern in Microdown

Microdown, the lightweight markup language designed for Pharo documentation, also relies heavily on the Visitor Design Pattern. As Microdown documents consist of various elements like headers, paragraphs, code blocks, and figures, the Visitor

Pattern allows efficient traversal and manipulation of these elements. Different operations, such as rendering in HTML or extracting documentation structure, can be implemented as visitors, keeping the elements themselves focused on representing content.

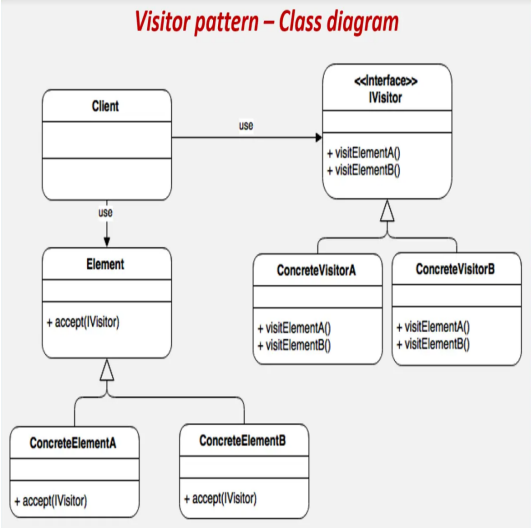

Figure 2.3: Visitor Design Pattern Structure

2.5.3.3 Implementation of Visitor in My Work

In my solution, I used the Visitor Pattern extensively to traverse and process various book elements. Each element, such as `BookTitle`, `AuthorInfo`, and `Chapter`, implemented the `accept` method, which allowed different visitor operations, such as validation or formatting, to be applied without modifying the underlying elements.

For example, to validate each element in the book, the visitor implementation invoked the correct operation depending on the type of the element being visited:

```
BookTitle>>accept: aVisitor
    aVisitor visitBookTitle: self.

Chapter>>accept: aVisitor
    aVisitor visitChapter: self.
```

This design made it easy to introduce new operations, such as exporting the book to different formats, without having to change the element classes themselves. The Visitor Pattern enabled a clear separation between the data structures (book elements) and the operations that could be performed on them.

2.5.3.4 Challenges Without the Visitor Pattern

Without the Visitor Pattern, adding new operations would require modifying each element class, violating the Open/Closed Principle (OCP), which states that software entities should be open for extension but closed for modification. The pattern provided the flexibility to extend the system with new operations, such as additional validation steps, without modifying the existing classes.

In conclusion, the Visitor Design Pattern significantly improved the maintainability and scalability of the book validation system, allowing for flexible and extendable operations across a diverse set of elements. Without this pattern, the system would have required extensive modifications for each new feature or operation, making it harder to maintain.

2.5.3.5 Pharo: A Modern, Open-Source Language

Pharo is a modern, open-source, dynamically-typed programming language that derives its inspiration from Smalltalk. It supports live coding, a practice that allows developers to write and modify code while the application is running, making it a powerful tool for rapid development and experimentation. Pharo is not only a language but also an entire ecosystem that emphasizes simplicity, dynamic execution, and continuous improvement [3].

The Pharo environment is composed of six key elements:

- **A dynamically-typed language** with a minimalist syntax, enabling users to write expressive, concise code. The syntax is so simple that it can be learned quickly and is readable even by those unfamiliar with it. It closely resembles natural language, reducing the cognitive load on the programmer.
- **Live coding environment:** Pharo allows for on-the-fly modification of code during execution, which helps developers test, refine, and experiment with features without restarting the application or recompiling the codebase.
- **Integrated Development Environment (IDE):** Pharo offers a powerful, integrated development environment designed to manage complex codebases while promoting good design principles like modularity and object orientation.
- **Rich library ecosystem:** Pharo's extensive libraries create a flexible development environment. It can be viewed as a virtual OS, with a fast Just-In-Time (JIT) compiler-based VM, and full access to system-level resources through its Foreign Function Interface (FFI).
- **Culture of innovation:** Pharo is built around a community that encourages contributions, rapid iteration, and continual improvement of both the language and the ecosystem.

- **Inclusive community:** Pharo has an active and welcoming global community, making it suitable for developers of all experience levels and backgrounds. The community fosters collaboration and is open to contributions from developers worldwide.

Pharo provides a robust platform for professional software development, as well as research and educational projects. Its design philosophy encourages incremental improvements over radical changes, leading to a system that is both stable and cutting-edge. Pharo's commitment to its open-source roots ensures that it remains free and accessible to everyone.

2.5.3.6 Why Pharo and Not Other Languages?

Pharo's unique combination of live coding, dynamic typing, and an object-oriented paradigm distinguishes it from other modern programming languages. Unlike many contemporary languages like Java, C++, or Python, Pharo's tight integration of the language, IDE, and development environment makes it a highly productive tool for software development.

Here are some reasons why Pharo stands out in comparison to other programming environments:

- **Live Coding:** Unlike languages like Java or C++, which require recompilation and restarting of applications after each change, Pharo allows developers to modify code in real-time as the application is running. This feature speeds up the development cycle significantly, as developers can see immediate results from their changes.
- **Simplicity and Minimalist Syntax:** Pharo's syntax is much simpler than many other object-oriented languages. It emphasizes readability and ease of learning, in contrast to more verbose languages like Java or C++, which have a steeper learning curve.
- **Full Object-Oriented Approach:** Everything in Pharo is an object, adhering strictly to the object-oriented paradigm. In contrast, languages like Python or JavaScript, while object-oriented, allow for procedural programming as well. Pharo's purist approach ensures that the code remains consistent and modular [12].
- **Rich Tooling:** The Pharo IDE is tightly coupled with the language, offering advanced features such as an interactive debugger, refactoring tools, and a class browser. Unlike other environments, where the IDE is often a separate tool, Pharo integrates all development tools within its environment.

- **Dynamic and Extensible:** Pharo allows for dynamic modification of objects and code at runtime, which is not a feature present in many statically-typed languages like C++ or even statically-typed functional languages like Haskell. This allows for highly flexible and adaptable code.
- **Focus on Educational and Research Purposes:** Pharo's origins in Smalltalk make it an excellent platform for learning object-oriented programming concepts and for conducting research on dynamic languages.

Feature	**Pharo**	**Java**
Type System	Dynamically typed	Statically typed
Live Coding Support	Yes	No (requires recompilation)
Object-Oriented	Fully object-oriented	Object-oriented with some procedural elements
Syntax Complexity	Minimalist, very simple	Verbose, requires extensive boilerplate code
Community Involvement	Open-source with active community contributions	Mainly used in corporate environments, less community-driven
Development Tools	Integrated with IDE, seamless debugging	IDEs (like Eclipse, IntelliJ) are separate from the language and VM
Cross-Platform	Fully portable on all major platforms (OS X, Linux, Windows, iOS, Android)	Cross-platform but often depends on JVM compatibility
Extensibility	Highly extensible with live modifications at runtime	Limited extensibility without recompilation or reflection

Table 2.2: Comparison between Pharo and Java

2.5.3.7 History and Evolution of Pharo

Pharo originated as a fork of Squeak, a modern implementation of the Smalltalk-80 language. Pharo was created in 2008 to refine Squeak's features while focusing on creating a more polished and professional-grade environment [3]. Over the years, Pharo has evolved to become one of the leading platforms for both software development and academic research in dynamic languages.

The first beta version of Pharo (1.0) was released in 2009. Since then, new versions have been consistently released, with major improvements to the virtual machine, language features, and integrated libraries. Pharo's latest version, 9.0, was released in July 2021. Each release introduces new features while maintaining the core principles of simplicity, object-orientation, and live coding.

Pharo is built with portability in mind. It runs on all major operating systems, including OS X, Windows, Linux, iOS, and Android. Its virtual machine is entirely written in a subset of Pharo, allowing developers to debug and modify the virtual machine itself using Pharo's tools. This level of introspection is rare in other languages and offers Pharo a unique advantage for both development and research purposes.

2.5.3.8 Pharo in the Context of My Work

In my work on the book validation system, Pharo played an instrumental role. The combination of live coding, dynamic typing, and the rich development environment allowed me to implement and test different strategies quickly and efficiently. By leveraging Pharo's extensibility and dynamic nature, I could design and apply multiple validation strategies to different sections of a book (such as title, content, and metadata) without significant overhead. The ability to modify and refine code in real-time provided the flexibility needed to handle complex validations seamlessly, something that would have been much harder to achieve in other environments.

Pharo's integration with the Strategy and Visitor Design Patterns made it easier to structure the validation operations and maintain the codebase. The flexibility of the language allowed me to iterate rapidly, testing different approaches without the cumbersome overhead often found in statically-typed languages.

In summary, Pharo provided the perfect balance of flexibility, speed, and power, allowing me to implement complex patterns and algorithms in a highly productive manner.

2.5.3.9 Pillar: A Versatile and Extensible Lightweight Markup Language

[1]

Pillar is a versatile, lightweight markup language that has become an essential tool within the Pharo ecosystem for creating structured documents such as technical books, websites, blogs, and presentations. Its flexibility comes from the ability to export content in various formats, including HTML, LaTeX, Markdown, and Asciidoc, from a single source file. This makes it an efficient tool for producing multi-format documentation with minimal effort, allowing authors to focus on content creation rather than output formatting.

Pillar was originally developed as a part of the Pharo ecosystem, which emphasizes simplicity and ease of use. It was specifically designed to allow Pharo developers to document their projects more effectively, while also catering to a broader audience of technical writers and content creators.

2.5.3.10 Key Features of Pillar

Pillar offers several features that make it stand out from other lightweight markup languages:

- **Simple and Consistent Syntax:** Pillar follows a minimalist approach, which ensures that documents are easy to read and write. The language uses a clean and consistent syntax that resembles popular markup languages like Markdown but extends them with additional features suitable for structured documents.
- **Multi-Format Export:** One of the key strengths of Pillar is its ability to generate multiple output formats from the same source file. Whether you need a web page (HTML), a printed book (PDF via LaTeX), or online documentation (Markdown), Pillar handles it seamlessly.
- **Support for Code Blocks and Annotations:** Pillar was designed with programmers in mind, so it includes robust support for code blocks, class definitions, and method definitions, which can be tested and verified. Annotations allow you to include advanced elements like screenshots, executable code, and more.
- **Extensibility:** Pillar's architecture is designed to be easily extendable. This allows developers to add custom features, such as new commands and annotations, to meet specific project needs.

2.5.3.11 The Compilation Chain of Pillar

The Pillar compilation chain takes the source file written in Pillar's syntax and transforms it into the desired output format. The process involves several stages,

ensuring that the document is correctly parsed, processed, and formatted according to the output type. The key steps in the compilation chain are:

1. **Parsing:** Pillar begins by parsing the source file, identifying the structure and elements such as headers, lists, code blocks, and annotations.

2. **Transformation:** Based on the desired output format, Pillar applies transformations to the parsed content. For example, in a LaTeX output, code blocks may be transformed into appropriately formatted environments, while in HTML output, they may become `<pre>` tags.

3. **Rendering:** Finally, the transformed content is rendered into the output format. This could involve converting the document into HTML for a website, or into LaTeX for a typeset PDF document.

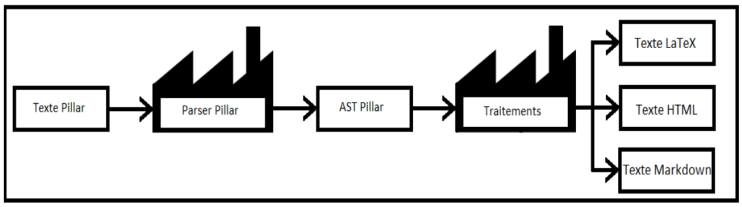

Figure 2.4: The Pillar Compilation Chain

Pillar also integrates with tools such as LaTeX, ensuring that documents can be rendered with professional-quality typesetting, suitable for academic publications, while also supporting lightweight formats for web use.

2.5.3.12 Pharo and Pillar: Why Pharo?

Pharo plays a central role in the development and execution of Pillar. Since Pharo itself encourages live coding and rapid iteration, it provides a conducive environment for refining documentation tools like Pillar. Pharo's extensibility and dynamic nature make it possible to implement custom features for Pillar, especially when dealing with complex outputs like LaTeX or HTML.

Pharo's lightweight and highly interactive development environment contrasts with more traditional, heavyweight tools used for typesetting and documentation in languages like LaTeX or Markdown. Pharo allows developers to experiment, modify, and immediately see results, which enhances the development process for tools like Pillar.

2.5.3.13 Concrete Use and New Additions

Pillar's design extends to use cases like book writing and project documentation. Testing code blocks is a notable feature, where Pillar integrates with the Pharo environment to check if the code examples included in the documentation are valid and execute correctly. This ensures that published books or websites contain working examples, a feature that is particularly valuable for technical publications.

In my work on the book validation system, Pillar was instrumental in generating the final documentation for the project. The integration with Pharo allowed me to dynamically validate code blocks within the documentation, ensuring consistency and accuracy across multiple output formats. Pillar also simplified the process of updating documentation as the project evolved, making it easier to maintain up-to-date technical content.

Feature	**Pillar**	**Markdown**
Syntax Complexity	Moderate (supports more features)	Simple (but limited in functionality)
Multi-Format Export	Yes (HTML, LaTeX, Markdown, Asciidoc)	Yes (HTML, LaTeX)
Code Block Testing	Supported (with validation)	Not natively supported
Extensibility	High (can be customized for advanced features)	Limited (with basic customization possible)

Table 2.3: Comparison between Pillar and Markdown

As a result, Pillar not only facilitated the generation of high-quality documentation but also improved the workflow by integrating testing and rendering into a unified process. This made it an indispensable tool for the book validation project and significantly contributed to its success.

2.5.3.14 Microdown: A Clean, Extensible Markup Language for Pharo Documentation

[7]

Microdown is a lightweight, extensible markup language specifically designed to meet the unique documentation needs of the Pharo ecosystem. It emerged as a response to the limitations of existing markup languages, particularly Markdown, which, while widely used, lacks certain advanced features necessary for writing structured and technical documents like books, class comments, and technical reports.

Microdown integrates seamlessly with Pharo, providing a unified syntax for all forms of documentation, from simple class comments to complex, multi-page technical documents. Its clean and consistent syntax, combined with its extensibility, makes it the go-to choice for developers and researchers working in the Pharo environment.

2.5.3.15 Key Features of Microdown

Microdown was designed to offer several improvements over existing markup languages, especially in the areas of consistency and extensibility. The main features of Microdown are:

- **Consistent Syntax:** Microdown addresses one of the key issues in Markdown—its inconsistent syntax. In Markdown, for example, headers can be defined in multiple ways, which can lead to confusion. Microdown eliminates such ambiguity by enforcing a consistent syntax across all document elements. This clarity ensures that developers, regardless of their familiarity with the language, can easily read and maintain documents written in Microdown.
- **Extensibility:** Microdown's design is inspired by Pillar's extension mechanisms, allowing developers to extend the language with custom annotations and environments. These extensions are marked using ‘‘ syntax, enabling users to incorporate elements like footnotes, citations, or even figures with specific parameters. This flexibility is crucial for Pharo users who need to document complex systems with varying levels of detail.
- **Support for Advanced Features:** Unlike Markdown, which lacks native support for structured elements like anchors or references, Microdown includes support for advanced document features. For instance, developers can use anchors to create internal references within a document, allowing easy navigation across sections. It also supports figure and code block parameters, making it suitable for technical documentation that requires embedded visual elements or complex code snippets.
- **Unified Syntax for Pharo Documentation:** Microdown serves as the standard markup language for Pharo documentation, unifying the way class comments, package comments, and external documents are written. This standardization reduces the cognitive load on developers, as they no longer need to switch between multiple languages or formats. The unification ensures that all documentation is consistent and maintainable within the Pharo ecosystem.

2.5.3.16 Why Microdown Matters for Pharo

Microdown's relevance to Pharo extends far beyond being a simple markup language. It plays a crucial role in improving the overall quality of documentation within the Pharo environment by streamlining the process of writing and maintaining technical documentation. Here are some of the key reasons why Microdown is important for Pharo:

- **Improving Class and Package Comments:** One of Microdown's primary uses is to enhance the quality of class and package comments within Pharo. With its clean syntax and support for rich text rendering, Microdown allows developers to write detailed, structured documentation that is directly embedded within the Pharo image. This improves the readability of codebases and encourages better documentation practices, ensuring that Pharo projects remain accessible and understandable even as they grow in complexity.
- **Seamless Integration with Pharo Tools:** Microdown's native integration with Pharo's development tools is a major advantage. Documentation written in Microdown can include live hyperlinks to classes or methods, which makes it easier for developers to navigate the codebase. Moreover, the integration allows for real-time rendering of documentation, so developers can immediately see the impact of their changes, whether they are working on a small class comment or a large technical report.
- **Cross-Platform Documentation:** Microdown was designed with cross-platform compatibility in mind. Documents written in Microdown can be exported into multiple formats, including HTML and LaTeX, without losing their structure. This ensures that documentation created within Pharo can be easily shared or published outside the Pharo environment, whether as online documentation, printed books, or technical reports. Microdown's flexibility ensures that Pharo projects can reach a wider audience without the need for extensive reformatting or rewriting.
- **Real-Time Feedback and Live Coding:** Thanks to Pharo's live coding environment, developers can write and modify their documentation as they work on their code. Microdown supports this by allowing real-time updates to documentation without the need for recompilation. This feature is particularly useful for teams working on large projects, as it ensures that documentation is always up-to-date with the current state of the codebase.

2.5.3.17 The Pillar Compilation Chain and Microdown

Microdown is integrated into the broader Pillar toolchain, which is used within the Pharo ecosystem for converting structured documents into a variety of formats,

including HTML, LaTeX, Markdown, and others. This integration makes it possible to write documentation once and export it into multiple formats, simplifying the process of creating and maintaining both internal and external documentation.

The Pillar compilation chain works in the following steps:

1. **Document Creation:** The developer writes the document in Microdown, which serves as the source file. Microdown's simple and clean syntax allows for the creation of both simple and complex documents with ease.

2. **Parsing and Conversion:** Pillar then parses the Microdown document and converts it into an abstract syntax tree (AST). The AST serves as an intermediate representation of the document, which can be manipulated and transformed as needed.

3. **Format Generation:** Once the document is parsed, Pillar generates the final output in the desired format, whether that's HTML for web-based documentation, LaTeX for academic publications, or Markdown for simple text-based formats.

The integration of Microdown into Pillar ensures that Pharo developers have a consistent and efficient way to create structured, high-quality documentation that can be shared across multiple platforms without losing its formatting or structure.

Intégration de Microdown dans la chaîne de compilation de Pillar

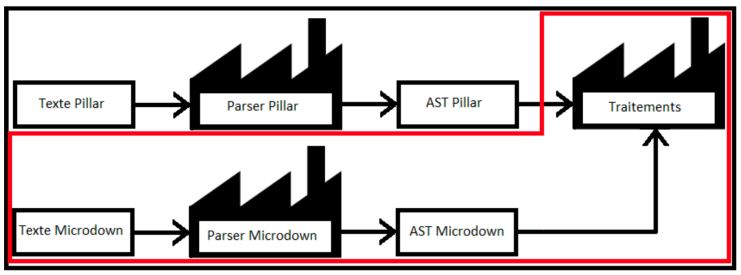

Figure 2.5: Pillar Toolchain with Microdown Integration

2.5.3.18 Comparison of Microdown with Markdown and Pillar

The following table illustrates how Microdown compares to standard Markdown and Pillar in terms of features and use cases:

Feature	Microdown	Markdown	Pillar
Extensibility	High, supports custom annotations and environments	Limited, lacks native support for advanced features	High, supports advanced extensions like footnotes, citations
Figures and Anchors	Supports figure parameters, anchors, and references	Limited support for figures, no anchors	Full support for figures, citations, and references
Syntax Consistency	Enforces a single syntax for elements	Multiple syntaxes for the same elements	Consistent syntax across all document types
Output Formats	HTML, LaTeX, Markdown (via Pillar)	HTML, limited LaTeX support	HTML, LaTeX, PDF, Markdown
Integration with Pharo	Fully integrated for class comments and documentation	Not integrated	Full support for Pharo documentation, books, and technical reports

Table 2.4: Comparison between Microdown, Markdown, and Pillar

2.5.3.19 Microdown in My Work

In my work on the book validation system, Microdown played a critical role in ensuring that the documentation was both well-structured and easy to maintain. The ability to define custom annotations and environments allowed me to document complex validation strategies, ensuring that each step of the validation process was clearly explained and easy to follow. Additionally, Microdown's integration with the Pillar toolchain made it possible to export the documentation into multiple formats, allowing me to easily share my work with both technical and non-technical stakeholders.

The seamless integration between Microdown and Pharo's development tools also allowed me to keep the documentation in sync with the system as it evolved. Since the documentation could be updated in real-time, there was never a disconnect between the state of the code and the documentation, ensuring that all team members had access to the most up-to-date information.

2.5.3.20 Conclusion

Microdown provides Pharo developers with a powerful, flexible, and extensible tool for writing high-quality documentation. Its clean and consistent syntax, combined with its seamless integration into the Pillar toolchain, ensures that Pharo

projects can maintain clear and well-structured documentation, regardless of their complexity. Whether used for writing class comments or full technical reports, Microdown is an essential part of the Pharo documentation ecosystem, enabling developers to write once and export their work across

CHAPTER 3

BOOK VALIDATION CHALLENGES

3.1 Structural Problems in Micro-down Documentation

3.1.1 Overview

One of the major challenges in handling large-scale documentation, especially when spread across multiple files, lies in ensuring that the document structure remains valid and intact throughout various linked sections. In the context of Microdown, which supports modular documentation via `inputFile` references (allowing sections to be split into separate files), structural problems such as missing files, cyclic references, and incomplete file collections can arise. These issues not only hinder the proper rendering of documentation but also make it difficult for users to navigate and comprehend the information.

The Validation Tool addresses these structural problems by systematically verifying that all referenced files are present, preventing infinite loops in file references, and ensuring that all necessary files are included before validation begins. This is done using the `MicFilesCollector` and other associated testing mechanisms to detect and prevent these structural issues.

3.1.2 Missing Input Files

One of the most common issues that arise in modular documentation is the use of `inputFile` references to include content from external files. In Microdown, a file might include a reference such as:

```
# Section1
<!inputFile|path=sections/section2.md!>
```

This indicates that the content of `sections/section2.md` should be included in the current document. However, if `section2.md` does not exist, the document will be incomplete or break during rendering. This problem can be particularly difficult to track manually in larger projects where many files are interdependent.

The Validation Tool solves this by implementing a process that collects all referenced files before validation begins. The `MicFilesCollector` class specifically handles the detection of missing files by traversing through all `inputFile` references and reporting any file that does not exist in the project.

3.1.3 Cyclic File References

Another structural problem is the possibility of cyclic file references, where two or more files reference each other in a loop. For example:

- `section1.md` references `section2.md`
- `section2.md` references `section1.md`

This creates an infinite loop, making it impossible to correctly parse or render the document. Without proper handling, this can cause the documentation generation process to become stuck, repeatedly including the same files without resolving.

The solution to this problem lies in the visited set maintained by the `MicFilesCollector` class. As files are visited during the collection process, they are marked as "visited," preventing them from being processed again.

3.1.4 Anchor and Reference Problems

3.1.4.1 Duplicated Anchors

In Microdown, anchors are used to create internal references within documents, allowing readers to easily navigate between sections. For example:

```
# Section 1
@anchorSection1
```

Each anchor should be unique within the document to avoid confusion and ensure that references point to the correct section. However, in large documents or projects with multiple files, it's possible to accidentally define the same anchor in different sections or across files.

3.1.4.2 Unknown Anchors

In Microdown, unknown anchors occur when a reference is made to an anchor that does not exist within the document. Anchors serve as internal links to specific sections, and when an anchor is referenced but hasn't been defined, the link will not function properly, leading to navigation issues. This often happens in large or complex documents, or across multiple files, when the defined anchors are missing or incorrectly named, resulting in broken references.

For example:

```
# Section 1
@anchorSection1

...

See [Section 2]{*anchorSection2*}
```

In this case, if `@anchorSection2` is not defined anywhere in the document, the link to Section 2 will not work properly, causing a broken reference.

3.1.4.3 Anchor References in Mathematical Expressions, Figures, and Code

When we refer to an 'anchor,' we are also including mathematical expressions, figures, and code elements, as they all follow the same mechanism of being referenced by defining a specific anchor point.

3.2 Semantic Problems in Micro-down Documentation

3.2.1 Incorrect Code Block Evaluation

The primary semantic challenge in Microdown documentation occurs when code blocks that are meant to demonstrate specific concepts or computations produce incorrect results. In technical documentation, such as tutorials, guides, or reference materials, code examples play a vital role in illustrating how a particular method, function, or algorithm works. If a code block returns the wrong result, it can mislead readers and cause significant confusion.

For instance, consider a simple arithmetic expression like:

```
3 + 7 >>> 11
```

This is a straightforward example where the code block is expected to demonstrate the addition of two numbers, producing the result 10. However, if the expected result is mistakenly written as 11, it introduces a semantic error.

3.2.2 Logical Errors and Inconsistencies

In addition to simple evaluation errors, more complex logical errors can occur when code blocks contain expressions that are fundamentally flawed. Logical errors arise when invalid operations are performed, such as trying to add incompatible types (e.g., adding a number to a string). These errors aren't always immediately obvious and can go unnoticed without a mechanism in place to evaluate the code.

For example:

```
3 + "12" >>> 15
```

Here, the document might present this as a valid expression, but semantically, it's incorrect. The operation of adding a number to a string is not valid in most programming languages, and attempting to evaluate this would raise an error. Semantic validation detects these errors and ensures that they are corrected before the document is finalized.

3.2.3 Handling Exceptions in Code Blocks

Another critical aspect of semantic validation is ensuring that exceptions raised during the evaluation of code blocks are handled properly and reported to the user in a meaningful way. In many cases, the code within a document might fail due to runtime errors, such as:

- Attempting to divide by zero.
- Accessing an undefined variable.
- Calling a method that does not exist.

For example:

```
result = 10 / 0 >>> Infinity
```

In most programming environments, dividing by zero raises an error or an exception, as this operation is mathematically undefined. Without proper error reporting, the documentation might fail silently or provide incorrect information to the reader. Semantic validation catches these exceptions, reporting back to the user with clear and detailed error messages.

3.2.4 Handling Expected Failures

Not all failures in code are unintended. In some cases, a code block is expected to fail, and the failure serves an instructional purpose. For example, an author might want to demonstrate what happens when invalid input is provided to a function. These cases are marked as expected failures.

For instance:

````
```example=true&expectedFailure=true
10 / 0 >>> Infinity
````

In this case, the failure is deliberate, and the tool must handle it accordingly. Rather than reporting it as an error, the tool confirms that the failure was expected and marks the test as successful.

3.2.5 Ambiguous or Undefined Code

Sometimes, code blocks in a document might be **ambiguous** or **poorly defined**, leading to unexpected behavior during evaluation. For example, an incomplete or syntactically incorrect code block like:

```
3 +
```

This expression is incomplete and cannot be evaluated meaningfully. Without semantic validation, such errors might go unnoticed until a reader encounters them. Semantic validation identifies these incomplete blocks, alerts the author, and ensures that all code is fully defined and executable.

3.2.6 Semantic Integrity Across Multiple Blocks

In large documents, ensuring semantic consistency across multiple interrelated code blocks is crucial. For example, if a variable is defined in one block and referenced in another, semantic validation ensures that the relationship between these blocks remains valid and consistent.

Consider:

```
var = 10

var * 2 >>> 20
```

In this case, if `var` is defined in one block, it must be available in the subsequent block. Any inconsistency, such as a missing or incorrectly referenced variable, leads to semantic errors. Semantic validation maintains these relationships, ensuring the logical flow of the document remains intact.
```

3.3 Evolution Problems in Micro-down Documentation

3.3.1 Overview

The evolution problem in Microdown documentation arises when certain elements or code blocks evolve over time, and the changes need to be synchronized across all versions of the document. This challenge is particularly relevant when the documentation refers to evolving code, such as method definitions, class structures, or algorithms. If a method or definition is updated in one place but not reflected elsewhere, it leads to inconsistencies and confusion. The primary focus of the evolution problem is ensuring that definitions, methods, or code blocks remain in sync as both the document and the underlying codebase evolve.

3.3.2 Code Block Synchronization

A significant issue arises when a code block refers to a method or function that has changed in a newer version of the document or codebase, but the document does not reflect this update. For instance, if method A is defined in an earlier version of the document and later updated in the codebase, the document must be updated to reflect the latest version of the method.

If synchronization does not occur, the document becomes outdated and potentially misleading. The key challenge here is ensuring that any references to code blocks remain synchronized with the latest versions in the codebase.

3.3.3 Synchronization Tags in Documentation

To address the synchronization issue, Microdown uses synchronization tags (e.g., `sync=true`) in code blocks. These tags help identify whether a code block needs to be synchronized with the corresponding method or function in the codebase.

If the code block and the method are out of sync, the tool flags it as a desynchronization error, alerting the author to update the document. This helps prevent confusion caused by discrepancies between the codebase and the document.

3.3.4 Handling Broken Synchronization Declarations

Sometimes, the synchronization declaration itself can be incorrect or improperly formatted. For example, the `sync=true` declaration may be used, but the corresponding method in the codebase may be missing, or the origin specified in the document may be wrong. This results in a broken sync definition.

Common cases include:

- The class or method referenced in the sync tag doesn't exist.
- The syntax of the sync declaration is incorrect.
- The method name or class is misspelled in the document.

When these issues occur, the validation tool reports them so the author can correct the document and maintain consistency with the codebase.

3.3.5 Versioning and Method Evolution

Another challenge is handling versioning. As the code evolves over time, the document must evolve with it. If a method changes between versions, the documentation must be updated accordingly to reflect the latest state of the code. Failure to do so introduces versioning issues, where the document refers to an outdated method or process.

For example, if `simpleCode` in version 1.0 performs a specific calculation and is updated in version 2.0 to include additional logic, the document must reflect this change. If not, the document could mislead readers by presenting outdated information that no longer aligns with the current codebase or implementation.

3.3.6 Detecting Desynchronized Code

The tool detects desynchronization between the document and the actual codebase by comparing the content of code blocks with the current source code. If a discrepancy is found, the tool reports it as a desynchronization issue, allowing the author to update the document.

This detection mechanism is critical to ensuring that the document stays aligned with the codebase, preventing outdated or incorrect information from being presented to users.

3.3.7 Inconsistent References Across Versions

Sometimes, a document may reference multiple versions of a method or class, leading to inconsistent references. For example, one section of the document may refer to an old version of `simpleCode`, while another refers to the updated version. This can confuse readers, especially if both versions are presented as valid.

Ensuring consistency across the entire document is key to preventing such issues.

3.4 Conclusion

In summary, the key challenges in validating Microdown documentation can be grouped into the following categories:

- **Structural Problems:**
  - **Missing Input Files:** The document may reference files that are missing, causing incomplete rendering.
  - **Cyclic File References:** Files referencing each other in a loop can cause infinite loops during document generation.
  - **Anchor Problems:** Duplicated or undefined anchors can cause navigation issues within large or modular documents.
- **Semantic Problems:**
  - **Incorrect Code Block Evaluation:** Code blocks that produce incorrect results can mislead readers and create confusion.
  - **Logical Errors:** Invalid operations, such as adding incompatible types, may go unnoticed without proper validation.
  - **Handling Exceptions:** Proper reporting of runtime errors, such as division by zero or undefined variables, ensures clear feedback.
  - **Expected Failures:** Some failures are instructional, and the tool must recognize and handle them correctly.
  - **Ambiguous or Undefined Code:** Incomplete or poorly defined code blocks can lead to unexpected behavior.
  - **Semantic Integrity Across Multiple Blocks:** Ensuring that variables or methods referenced in one block are correctly defined in another is crucial for consistency.
- **Evolution Problems:**
  - **Code Block Synchronization:** When methods or functions evolve, the corresponding code blocks in the document must be updated to stay synchronized.
  - **Synchronization Tags:** Tags like `sync=true` help identify code blocks that require synchronization with the codebase.
  - **Handling Broken Synchronization:** Incorrect or outdated sync declarations can lead to broken references that must be reported and corrected.

  - **Versioning and Method Evolution:** As the code evolves across versions, the document must reflect these changes to prevent outdated information.
  - **Detecting Desynchronized Code:** The tool compares the document's code blocks with the source code, detecting any discrepancies.
  - **Inconsistent References Across Versions:** Multiple references to different versions of the same method can confuse readers, requiring consistent updates across the document.

Addressing these challenges ensures that Microdown documentation remains valid, consistent, and up to date, enhancing both readability and functionality for users.

3.5 Testing in the Book Validation System

In the context of the book validation system, testing plays a critical role in ensuring that all edge cases, exceptions, and various document structures are correctly handled. The book validation process involves performing both structural and semantic checks on Microdown documentation. With complex, modular documents, it is crucial to rigorously test every part of the system to guarantee that no errors or oversights occur during validation.

3.5.1 Importance of Testing in Book Validation

The book validation system must handle complex document structures spread across multiple files. To ensure this is done correctly, tests are vital for verifying that the system behaves as expected under various conditions. Here are the primary reasons why thorough testing is necessary in the context of book validation:

- **Handling Complex Document Structures:** When dealing with large-scale, modular documentation, missing files, cyclic references, and incorrect code block evaluations can disrupt the validation process. Tests ensure that all these components are correctly detected and handled.
- **Avoiding Breakages:** Documentation evolves over time, and changes can introduce new issues. Automated tests help ensure that any change or update to the system does not break existing functionality.
- **Consistency Across Versions:** Tests maintain consistency across multiple versions of a document or codebase. In the case of evolving code blocks or method definitions, testing ensures that the corresponding documentation remains in sync.

- **Edge Case Coverage:** In complex systems, corner cases or unexpected situations are common. Tests help cover all edge cases that might otherwise go unnoticed during regular development.
- **Ensuring Accuracy of Validation:** Semantic issues, such as undefined anchors or logical errors in code blocks, need to be identified and corrected. Testing helps ensure that validation is accurate and catches all potential issues.

3.5.2 Test-Driven Development (TDD)

For the book validation system, a test-driven development (TDD) approach was adopted to ensure robust and reliable code. TDD is a software development methodology where tests are written before the actual code is implemented. By following this methodology, every feature in the validation system is accompanied by corresponding test cases, ensuring that the system functions as expected from the outset.

The key steps in TDD are as follows:

1. **Write a Test:** Before writing any code, a test case is created that specifies the desired behavior of a feature. For instance, a test might verify that the system can detect a missing file or identify a cyclic reference in the document.
2. **Run the Test:** Initially, the test will fail because the code to implement the feature has not yet been written. This failure confirms that the test is correctly identifying the absence of the desired behavior.
3. **Implement the Code:** The next step is to write the minimal amount of code needed to pass the test. At this stage, the implementation focuses solely on fulfilling the specific requirements of the test.
4. **Refactor the Code:** After passing the test, the code is refactored to improve its design and ensure maintainability, without altering its behavior. The refactoring process should still ensure that the test case passes.
5. **Repeat:** The process repeats for each new feature or functionality, gradually building up the system with fully tested, reliable code.

3.5.3 Extreme TDD in Book Validation

In this project, the principles of Extreme Test-Driven Development (Extreme TDD) were applied. This methodology goes beyond standard TDD by emphasizing even smaller, incremental changes and more frequent testing. The idea is to

break down each feature into the smallest possible components and test each one thoroughly before moving on to the next.

Extreme TDD ensures:

- **Immediate Feedback:** Testing very frequently provides immediate feedback on whether the code works as expected.
- **Granular Test Coverage:** By breaking down features into smaller components, Extreme TDD ensures that every part of the system is covered by tests.
- **Minimized Debugging Efforts:** Errors are caught early and fixed immediately, reducing the need for extensive debugging later in the development process.

3.5.4 Writing Tests for Book Validation

In the book validation system, tests are written for various structural and semantic validation checks. The following types of tests were included:

- **Structural Validation Tests:** These tests ensure that the document structure is intact, checking for issues like missing input files, cyclic file references, and duplicated anchors.
- **Semantic Validation Tests:** Semantic tests verify the correctness of code block evaluations, ensuring that no logical errors or incorrect outputs occur within the document.
- **Edge Case Tests:** Special cases, such as handling invalid input or unknown anchors, are thoroughly tested to ensure that the system can gracefully handle any unexpected scenarios.
- **Evolution Tests:** As the document evolves, tests ensure that changes are synchronized across multiple files and that any modifications to code blocks are correctly reflected throughout the document.

3.5.4.1 Example Test Case in Pharo

Below is an example of a test case written in Pharo to check for missing input files in the book validation system:

```
testMissingInputFile
   | document |
   document := BookValidator new.
```

```
self should: [document validate: 'bookWithMissingFile.md']
   raise: MissingFileException.
```

This test case creates a new instance of the `BookValidator` class and verifies that attempting to validate a document with a missing file raises the appropriate exception. This ensures that missing files are detected correctly during the validation process.

3.5.5 Conclusion

In conclusion, testing is an essential aspect of the book validation system, ensuring that it can handle the structural and semantic challenges associated with large-scale documentation. By adopting Test-Driven Development (TDD) and Extreme TDD, the system was built incrementally, with tests driving the implementation of each feature. This approach guarantees that every part of the system is thoroughly tested and that potential issues are caught early in the development process. Testing not only improves the reliability of the system but also enhances its maintainability and scalability over time.

CHAPTER 4
SYSTEM DESIGN & ARCHITECTURE

4.1 System Design

The system design for the book validation tool is structured around solving the key challenges discussed in the previous chapter. The design aims to ensure that the document structure remains valid, semantic errors are detected early, and evolution issues are handled efficiently as the documentation changes. This section presents the design decisions and approaches taken to solve each of the structural and semantic challenges encountered in Microdown documentation validation.

4.2 Implementation

4.2.1 Cyclic File References

Cyclic file references occur when two or more files reference each other in a loop, creating an infinite recursive inclusion. This can prevent the document from being rendered properly, as the system will continue to reference the same files over and over again. The solution to this problem is to break the cycle by keeping track of files that have already been visited during the validation process.

4.2.1.1 Design Approach: Preventing Infinite Loops

To avoid infinite loops, the `MicFileCollector` maintains a visited set of files. When a file is encountered for the first time, it is added to the set. Any subsequent

attempts to visit the same file are ignored, effectively breaking the cycle and ensuring that each file is processed only once.

```
visited := Set new.
[ worklist isEmpty ] whileFalse: [
    currentDocument := worklist removeFirst.
    visited add: currentDocument fromFile.
]
```

This approach prevents cyclic references from disrupting the validation process and ensures that all files are processed correctly.

4.2.2 Anchor and Reference Problems

Anchors are critical for navigation within large documents, as they allow sections to be linked and referenced internally. However, several issues can arise with anchors, such as duplicated anchors or references to undefined anchors. These problems can lead to navigation errors, making it difficult for users to move between sections in the document.

4.2.2.1 Design Approach: Detecting and Reporting Anchor Issues

To handle anchor and reference problems, the system implements the following checks:

- **Duplicate Anchors:** The system ensures that each anchor within a document is unique. If two or more anchors are found with the same identifier, the system flags this as an error and reports it to the user.
- **Undefined Anchors:** If a reference is made to an anchor that does not exist, the system flags this as an undefined anchor. This helps to ensure that all links within the document are valid and that no broken references exist.

```
self anchorResults do: [ :anchorResult |
    anchorResult anchorLabel ifNil: [
        self undefinedAnchors add: anchorResult ].
]
```

The validation tool uses the `MicAnchorResult` class to track and report issues related to anchors.

4.2.3 Semantic Problems in Documentation

The validation tool also handles semantic issues that arise when code blocks or other document elements contain incorrect or invalid content. These issues are particularly common in technical documentation where code examples are used to illustrate specific concepts or computations.

4.2.3.1 Incorrect Code Block Evaluation

One common semantic problem is incorrect code block evaluation. This occurs when a code block that is supposed to demonstrate a specific concept produces an incorrect result. For example, a simple arithmetic expression like `3 + 7` might be expected to produce the result 10, but if the documentation mistakenly shows the result as 11, this would be flagged as a semantic error.

Design Approach: Code Block Testing

The system ensures the accuracy of code blocks by executing them during the validation process. It compares the actual output of the code with the expected output specified in the documentation.

```
self testBlock: '3+4' expected: '7'.
```

If the two results differ, the system flags the code block as incorrect and reports the error to the user.

4.2.3.2 Logical Errors and Inconsistencies

The tool simulates the execution of code blocks and checks for logical inconsistencies. If a code block contains invalid operations (e.g., trying to add a number to a string), the system generates an appropriate error message.

```
[ 3 + '12' ] on: Error do: [ :ex |
    self log: 'Logical error detected: ', ex description.
].
```

4.2.4 Handling Exceptions in Code Blocks

Exceptions can occur during the evaluation of code blocks, especially when invalid operations are performed. Common exceptions include division by zero, accessing undefined variables, or calling methods that do not exist. These exceptions need to be handled gracefully to ensure that the document validation process is not disrupted.

4.2.4.1 Design Approach: Exception Handling

To handle exceptions, the system executes code blocks in a controlled environment and catches any exceptions that are raised.

```
[ result := 10 / 0 ]
    on: ZeroDivide do: [ :ex |
    self handleException: ex.
].
```

The `MicBookTestResult` class manages the results of code block evaluations, including any exceptions raised during execution.

4.2.5 Handling Expected Failures

Not all failures in code blocks are unintended. In some cases, a failure is expected and serves an instructional purpose. For example, a documentation author may want to demonstrate what happens when invalid input is provided to a function. These cases are marked as expected failures.

4.2.5.1 Design Approach: Expected Failures

The system allows authors to mark certain code blocks as expected failures using special annotations.

````
/```example=true&expectedFailure=true
10 / 0 >>> Infinity
/```
````

If the block fails as expected, the system records the result but does not report it as an error.

4.2.6 Semantic Integrity Across Multiple Blocks

In large documents, it is important to ensure that the relationships between code blocks remain consistent. For example, a variable defined in one block should be available in subsequent blocks that reference it. Any inconsistencies between code blocks can lead to errors or incorrect behavior.

4.2.6.1 Design Approach: Ensuring Semantic Consistency

The system tracks the definitions and references of variables, methods, and other elements across multiple code blocks.

```
variables at: 'var' put: 10.
result := (variables at: 'var') * 2.
self assert: result equals: 20.
```

If a reference is made to an undefined variable or method, the system flags this as an error.

4.2.7 Evolution Problems in Documentation

As documentation evolves, certain elements or code blocks may change over time, and these changes need to be synchronized across all versions of the document.

4.2.7.1 Design Approach: Code Block Synchronization

The system uses synchronization tags (`sync=true`) to track which code blocks need to be synchronized with corresponding elements in the codebase.

```
sync=methodX
self syncCodeBlock: 'methodX'.
```

If a discrepancy is found, the system reports the issue and prompts the user to update the document.

4.2.8 Versioning and Method Evolution

Handling versioning issues is critical when a document refers to evolving code. If a method changes between versions, the document must be updated accordingly.

4.2.8.1 Design Approach: Version Control Integration

The validation tool integrates with version control systems to track changes in the codebase and ensure that the documentation stays up-to-date.

```
if (versionControl currentVersion differsFrom: documentVersion) [
    self flagAsOutdated: 'methodX'.
].
```

Problem	Solution Approach
Cyclic File References	Use a `visited` set to track already-processed files and prevent infinite loops.
Anchor and Reference Problems	Ensure anchors are unique and check for undefined references using `MicAnchorResult`.
Incorrect Code Block Evaluation	Execute code blocks and compare actual vs. expected results using `MicBookTestResult`.
Logical Errors and Inconsistencies	Detect invalid operations during code execution and log errors.
Handling Exceptions in Code Blocks	Catch exceptions during code block execution and handle gracefully using `on:do:` blocks.
Handling Expected Failures	Mark certain blocks as `expectedFailure` and skip error reporting for those cases.
Semantic Integrity Across Blocks	Track variable and method definitions across multiple blocks to ensure consistency.
Code Block Synchronization	Use `sync=true` tags to ensure code blocks remain synchronized with the latest codebase changes.
Versioning and Method Evolution	Integrate with version control to track changes and flag outdated references.

Table 4.1: Summary of Solutions to Documentation Validation Challenges

4.3 Summary of Approaches

4.4 Conclusion

The implementation of the book validation tool addresses the key structural and semantic challenges discussed in the previous chapter.

CHAPTER 5

USER GUIDE

This chapter provides a comprehensive user guide for utilizing the Microdown validation checker, integrated within the Pillar toolchain. The tool is designed to help authors and developers identify and resolve structural issues, missing references, and undefined anchors in their documentation. By following the instructions and workflow outlined in this chapter, users can ensure that their documents are free of errors, consistent, and up to the quality standard required for technical publications.

5.1 Installing and Setting Up the Toolchain

To begin using the Microdown checker integrated with Pillar, you need to clone and set up the Pillar repository on your local machine. Follow these steps:

1. Clone the Pillar repository:

```
git clone https://github.com/pillar-markup/pillar.git
```

2. Navigate to the cloned repository and check out the desired version:

```
cd pillar
git checkout v10.3.0
```

3. Set the necessary execution permissions and build the tool:

```
chmod a+x ./scripts/build.sh
./scripts/build.sh
```

4. Clone an example booklet to test the setup, such as the Pharo Virtual Machine booklet:

```
git clone https://github.com/SquareBracketAssociates/Booklet-PharoVirtualMachi
```

At this stage, you have successfully set up the Pillar toolchain and can begin validating your documentation.

5.2 Running the Reference Checker

Once the toolchain is set up, you can use the 'pillar referenceCheck' command to check for structural issues, undefined anchors, and broken references in your documentation.

To run the reference checker, use the following command:

```
pillar referenceCheck index.md
```

Where `index.md` is the root document of your project.

5.2.1 Example Output

After running the command, the tool will report any issues it finds. Below is an example output:

```
/Users/ducasse/Documents/Pharo/vms/120-x64/Pharo.app/Contents/MacOS/Pharo
/Users/ducasse/Test2/pillar/build/Pharo.image clap pillar referenceCheck index.md

File index.md has reference problems.

## Duplicated Anchors:

Anchor sec:references is duplicated in file: /Users/ducasse/Workspace/FirstCircle/
MyBooks/Bk-Writing/PharoBooks2/Booklet-PharoVirtualMachine/Part0-Preamble/1-ObjectStruc

## Undefined Anchors:

Anchor fig:classes is undefined in file: /Users/ducasse/Workspace/FirstCircle/
MyBooks/Bk-Writing/PharoBooks2/Booklet-PharoVirtualMachine/Part0-Preamble/1-ObjectStruc
```

The tool outputs a list of problems it encountered in the document, categorized into:

- **Duplicated Anchors:** These occur when the same anchor is defined more than once within the same document or across different files.
- **Undefined Anchors:** These occur when an anchor is referenced in the document but has not been defined.

5.3 Resolving Common Issues

5.3.1 Duplicated Anchors

Duplicated anchors are problematic because they confuse internal navigation within the document. To resolve duplicated anchors:

- Identify where the anchor is duplicated by looking at the file paths provided by the tool.
- Modify the anchor labels to ensure they are unique within the document.

5.3.2 Undefined Anchors

Undefined anchors lead to broken references in the document, where a link points to a non-existent section or figure. To resolve this:

- Locate the reference to the undefined anchor using the file path provided in the output.
- Either define the missing anchor or remove the reference if it is no longer needed.

5.4 Handling File References

When working with modular documentation, it is common to link external files using `inputFile` tags. The checker will also validate these file references. For example:

```
<!inputFile|path=sections/section2.md!>
```

If a referenced file is missing, the checker will report it. To resolve:

- Ensure that all referenced files are correctly named and exist in the specified paths.

5.5 Integration with CI/CD Pipelines

The reference checker can be integrated into continuous integration (CI) or continuous deployment (CD) pipelines to automate validation of documentation. This ensures that no broken references are introduced into the documentation during the development process.

5.6 Conclusion

The Microdown reference checker, integrated with the Pillar toolchain, provides a robust solution for validating large-scale documentation projects. By following this guide, users can ensure that their documentation remains free from structural issues, duplicated or undefined anchors, and broken file references. The tool is flexible, extensible, and can be integrated with automated workflows to maintain high documentation quality across projects.

CONCLUSION

In this thesis, we explored the structural and semantic challenges inherent in validating large-scale Microdown documentation, and we presented a comprehensive system design to address these issues. The problems of missing input files, cyclic references, anchor inconsistencies, incorrect code block evaluation, and desynchronization between documentation and evolving codebases all pose significant risks to the integrity and usability of documentation systems.

By leveraging a well-structured validation tool, our solution efficiently handles these challenges. The `MicFilesCollector` ensures the document's structural soundness by detecting missing or cyclic references, while semantic validation helps catch logical and evaluation errors within code blocks. Additionally, our approach includes robust mechanisms to synchronize documentation with evolving codebases, ensuring consistency across versions and avoiding confusion caused by outdated references.

Testing has played a pivotal role in the development of this tool. Through Test-Driven Development (TDD) and Extreme TDD methodologies, we systematically ensured the correctness of the tool's functionalities. Testing not only provided early detection of issues but also strengthened the maintainability and scalability of the tool over time. The structured testing approach also allowed us to handle edge cases, ensuring reliability across diverse document structures and scenarios.

The user guide outlines how to set up and operate the validation tool, making it accessible to developers and authors working with modular documentation systems. Integrating this tool within a Continuous Integration (CI) pipeline ensures that documentation remains error-free and aligned with the latest code updates, thus enhancing the reliability and quality of the output.

In conclusion, the proposed book validation system provides a comprehensive and reliable solution for maintaining high-quality Microdown documentation. By addressing both structural and semantic issues, the system ensures that documentation remains accurate, navigable, and up to date, making it a valuable asset for

both technical writers and software developers in managing complex documentation projects. The system's flexibility, combined with its robust testing framework, offers a scalable solution capable of adapting to future documentation challenges.

BIBLIOGRAPHY

[1] Thibault Arloing, Yann Dubois, Stéphane Ducasse, and Damien Cassou. Pillar: A versatile and extensible lightweight markup language. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–5, 2016.

[2] John Baker. Challenges in software documentation: An agile perspective. *Journal of Software Engineering*, 9(2):45–53, 2015.

[3] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu. com, 2010.

[4] Robert Brown. Enhancing software documentation quality. In *Advances in Software Engineering*, pages 233–245. Springer, 2022.

[5] Aikaterini Christopoulou, Emmanouel A Giakoumakis, Vassilis E Zafeiris, and Vasiliki Soukara. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202–1214, 2012.

[6] David Davis and Ben Shneiderman. The importance of documentation in software development. *IEEE Software*, 25(4):78–82, 2008.

[7] Stéphane Ducasse, Laurine Dargaud, and Guillermo Polito. Microdown: a clean and extensible markup language to support pharo documentation. In *International Workshop of Smalltalk Technologies*, 2020.

[8] StŽphane Ducasse and Damien Pollet. *Learning Object-Oriented Programming, Design and TDD with Pharo*. Lulu. com, 2018.

[9] Maria Garcia. The importance of syntactic correctness in technical documentation. *Journal of Technical Communication*, 12(3):45–60, 2018.

[10] Ian Garside. *Software Documentation: Strategies and Techniques*. Springer, USA, 2012.

[11] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[12] Konrad Hinsen. 1. exploring pharo. 2018.

[13] Emily Johnson. Maintaining consistency in software documentation. *International Journal of Software Engineering*, 15(2):75–82, 2021.

[14] Robert Jones. Understanding technical documentation complexity in software development. In *Proceedings of the International Conference on Software Engineering*, pages 100–108, 2016.

[15] John Lee. Ensuring completeness and relevance in software documentation. In *Proceedings of the International Conference on Software Engineering*, pages 150–158, 2020.

[16] Tanumoy Pati and James H Hill. A survey report of enhancements to the visitor software design pattern. *Software: Practice and Experience*, 44(6):699–733, 2014.

[17] Alice Smith. *Effective Documentation: Strategies for Success*. Tech Press, USA, 2018.

[18] James Smith. *Technical Documentation: A Comprehensive Guide*. Tech Press, New York, USA, 2019.

[19] Emma Taylor. The role of subjectivity in software documentation quality. *Software Quality Journal*, 27(1):25–40, 2019.

[20] David A. Wheeler. *Understanding and Improving Software Documentation*. Prentice Hall, USA, 2006.