IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 April 2023, accepted 19 May 2023, date of publication 25 May 2023, date of current version 5 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3279823

RESEARCH ARTICLE

Automated Reusable Tests for Mitigating Secure Pattern Interpretation Errors

CARLOS CUNHA[1] AND NUNO POMBO[2], (Senior Member, IEEE)
[1]Polytechnic Institute of Viseu, Campus de Repeses, 3500-510 Viseu, Portugal
[2]Instituto de Telecomunicações, Universidade da Beira Interior, 6201-001 Covilhã, Portugal

Corresponding author: Carlos Cunha (cacunha@estgv.ipv.pt)

This work was supported in part by the National Funds through the FCT—Foundation for Science and Technology, I.P., under Project UIDB/05583/2020; in part by the Research Centre in Digital Services (CISeD) and the Polytechnic of Viseu; and in part by FCT/MCTES through National Funds and when applicable co-funded EU Funds under Project UIDB/EEA/50008/2020.

ABSTRACT The importance of software security has increased along with the number and severity of incidents in recent years. Security is a multidisciplinary aspect of the software development lifecycle, operation, and user utilization. Being a complex and specialized area of software engineering, it is often sidestepped in software development methodologies and processes. We address software security at the design level by adopting design patterns that encapsulate reusable solutions for recurring security problems. Design patterns can help development teams implement the best-proven solutions for a specialized problem domain. However, from the analysis of three secure pattern implementations by 70 junior programmers, we detected several structural errors resulting from their interpretation. We propose reusable unit testing test cases based on annotations to avoid secure pattern interpretation errors and provide an example for one popular secure pattern. Providing these test cases to the same group of programmers, they implemented the pattern without errors. The reason is annotations build a framework that disciplines programmers to incorporate secure patterns in their applications and ensure automatic testing.

INDEX TERMS Security, software design, software safety, software testing.

I. INTRODUCTION

Security is a non-functional software aspect with growing concern in recent years. A Gartner study predicted that by 2025, 45% of global organizations would be impacted by a supply chain attack [1]. Along with the number of attacks, many software vulnerabilities are growing. A report found that ethical hackers could discover over 65,000 vulnerabilities in 2022, up by 21% over 2021 [2].

Software vulnerability disclosures often originate at the architectural and design levels. In some specific areas, as in industrial control systems, most security threats have architectural flaws as the primary root cause [3]. Security design flaws, as weaknesses in the high-level design of a system, lead to code defects that expose the system to security threats.

Software defects are tangible effects of poor software quality caused often by design flaws [4]. Design flaws harm the quality of the software. They indicate inappropriate design practices and principles, making the system harder to understand, maintain, and evolve. Security flaws and software quality are correlated [5]. Treating structural quality improvement as an iterative process to achieve the optimal quality thresholds is essential to mitigate security risks [6].

Software design patterns are valuable assets to improve the software's structural quality and harness it with proven design solutions to security vulnerabilities. They document the problem, solution, and application context and provide hints and examples for implementation. However, their understandability and mapping to implementation may not be understood by programmers, and their implementation is error-prone due to technical debt in software projects [6].

This article identifies problems related to the interpretation and implementation of secure design patterns and proposes

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

a solution based on reusable test cases. We believe combining secure design patterns for documenting the best security patterns with approaches for validating their implementation can solve most security flaws at the architectural and design levels.

A. CONTRIBUTIONS

This article makes the following contributions:

- Identifies implementation errors made by junior programmers implementing secure patterns for the first time;
- Discuss the origin of secure pattern implementation errors;
- Proposes a reusable test-based solution to detect secure pattern implementation errors.

Our work's contributions point toward using test-driven evaluation to improve software quality and mitigate security flaws in consonance.

B. RESEARCH QUESTIONS

We answer the following research questions:

RQ1. What are the typical programmer errors when implementing secure design patterns?

RQ2. How to create reusable test cases dependent on secure patterns to detect their implementation flaws?

Implementing secure design patterns can be prone to errors, potentially compromising the application's security. Understanding the typical programmer errors when implementing secure design patterns is essential to mitigate vulnerabilities and enhance the overall security posture of software.

By creating reusable test cases dependent on secure patterns, researchers can systematically evaluate the implementation of these patterns in software systems. This approach not only assists in identifying flaws and vulnerabilities but also facilitates continuous improvement and validation of secure design patterns. Ultimately, it contributes to building more secure and resilient software applications, minimizing the risks associated with insecure implementations of these critical patterns.

C. STRUCTURE

This article is structured as follows. Section II presents the related work. Section III describes, delimits, and validates the problem addressed by this article. Section IV explains the reusable test implementations proposed to mitigate structural errors resulting from implementation use cases. Section V presents the conclusions.

II. RELATED WORK

The purpose of this section is to provide a comprehensive analysis of the existing research in the field, with a specific focus on the studies most pertinent to our research questions. We will examine the prior work's methodology, findings, and conclusions to contextualize our study and identify gaps in the existing knowledge.

Two primary methods exist for evaluating software architecture and design against quality attributes. The first technique is known as scenario-based architectural analysis, which involves generating a series of evaluation scenarios – often involving brainstorming workshops – based on the specific requirements of the evaluation. The second technique is metrics-based, which focuses on developing specific metrics that can be used to assess the software architecture. These metrics are designed to evaluate various aspects of the architecture and provide objective criteria for measuring its quality. In [7], the authors perform design inspection by exploring the potential for automating the application of inspection rules to make security analysis more efficient. Design models are represented as data flow diagrams (DFD), which support the automation of five model inspection guidelines for security. An empirical evaluation of the automated guidelines shows acceptable precision and recall, but the paper also highlights limitations in the guidelines themselves, such as overlaps and unclear rules. The authors of [8] presented an architecture security analysis approach using security scenarios and metrics. It formalizes attack scenarios and security metrics using the Object Constraint Language (OCL). Formal signatures enable the localization of attack scenarios based on signature matches and take measurements for security metrics while analyzing a target system. The approach has been validated using NIST security principles and attack scenarios defined in the CAPEC database. In [9], the authors generated a test template from a security design pattern using aspect-oriented programming to observe the internal application processing. As the test template is reusable, it enables the easy creation of tests for validating security design patterns. The authors designed a web system as a case study scenario for experimental purposes. Notwithstanding the security design pattern is validated in the implementation phase, the test is manually created from the test template, creating a new source of errors.

Selecting an appropriate design pattern for a software system is an important decision that requires careful consideration of various factors such as the system requirements, architecture, development team expertise, and existing codebase. In [10], the authors present a mechanism for deciding the appropriate secure design patterns for online application vulnerabilities. Secure design patterns prevent vulnerabilities from being accidentally introduced into code or reduce the effects of flaws. The work presented in [11] explores the selection of secure patterns using text categorization regarding the software requirements specification. An evaluation of the approach using e-commerce, social media, and desktop utility programs resulted in an accuracy of 81% and recall up to 69%. Finally, a security-by-design approach was proposed in [12]. The authors identified a list of possible security attacks on the smart metering systems and introduced secure design patterns recognized for these systems.

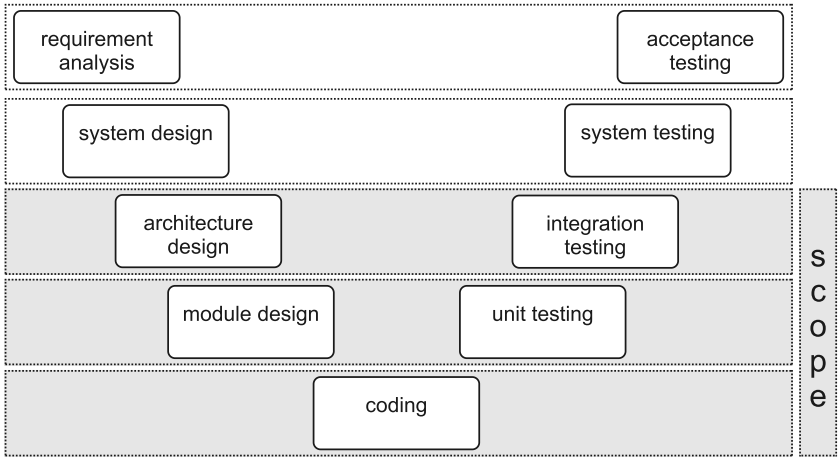

FIGURE 1. V-model methodology.

Previous work on secure software development based on architectural and design patterns focus on searching and appropriate patterns to solve a specific problem and on the security metrics to evaluate the software architecture. Secure software development needs more standardized approaches for development to reduce subjective per-project interpretation of the security properties. Design patterns can provide standard design solutions for secure software development, but mapping them to specific projects' designs is error-prone. Thus, their implementation can compromise the security features claimed for the design pattern. We evaluate the dimension of this problem by observing the implementation of several secure patterns by programmers. We also present a test-based approach that uses annotations as anchors to the structural application elements. With this approach, we aim to mitigate errors introduced by the interpretation and mapping from design patterns to the application design.

The use of annotations has been applied to the design of parallel applications [13], [14] and to support the application of concurrency patterns [15], but was never explored for pattern-oriented development of secure applications.

III. PROBLEM VALIDATION

This section describes, delimits, and validates the problem addressed by this article. Problem validation is based on the results obtained from implementing several secure design patterns by programmers. These results validate the research problem stated in this article.

A. PROBLEM DELIMITATION

Security design flows can be introduced at any phase of the Software Development Life Cycle (SDLC): requirement analysis, architecture and design, coding, testing, deployment, and production. Our work focuses on the software design and coding phases.

Architecture security risk analysis depends on the system architecture and design models. Existing efforts to evaluate software architecture against quality attributes are classified into:

- Metric-based approaches [16] applied to evaluate architectural security flaws using metrics adequate to the scope and applicability. These include static and dynamic vulnerability analysis, architecture security, and runtime security metrics.
- Scenario-based architectural analysis [17] using evaluation scenarios generated based on the requirements.

When adopting pattern-oriented software architectures, the reputation of the security pattern is built up from the peer-reviewing process in academia and its long-term reliability in real-world projects. Thus, the personalized per-project validation design inherent to metric and scenario-based approaches can be avoided. On the contrary, the mapping between each design pattern and its implementation is error-prone. The consequence of inappropriately applying a security pattern is not mitigating the target threats and vulnerabilities. Considering that the correctness of design patterns specification is a valid assumption, their implementation demands validation artifacts to avoid security flaws.

B. VALIDATION ARTIFACTS

Every development phase demands an associated validation phase. Figure 1 presents the V-Model [18], a popular model that represents the association of a testing phase with each corresponding development phase. This model is popular in the software testing community and was adopted by the ISO/IEC 25010-2011 [19], one leading standard for systems and software quality requirements and evaluation.

Unit and integration tests are artifacts used to validate the application at the architectural and module design levels. These tests can be done manually – e.g., code inspection sessions, walkthroughs [20] – or by resorting to automated software tests.

Automated testing offers several advantages over manual testing. It enables continuous integration and delivery using

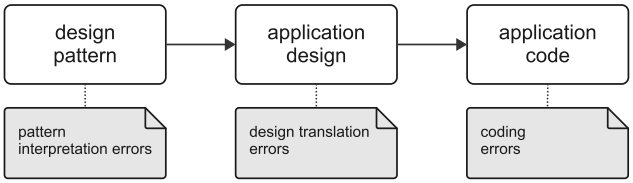

FIGURE 2. Errors types introduced at each stage of pattern implementation.

DevOps pipelines [21] and regression testing by reusing existing test cases during preventive, corrective, and evolving maintenance. Reusing test cases will contribute to better software quality and fewer security flaws since correct testing artifacts can be reused throughout several software versions.

C. PROBLEM VALIDATION

To answer the research question RQ1, we analyzed the code written by 70 junior programmers implementing three secure patterns to identify the errors resulting from translating each design pattern specification to the application domain. All programmers hold a bachelor's degree and are familiar with pattern-oriented programming.

Three error types can originate from pattern implementation (Figure 2):

1) Pattern interpretation errors resulting from the interpretation of the pattern documentation.
2) Design translation errors are introduced when restructuring the application design according to the pattern.
3) Coding errors caused by programmer mistakes and misinterpretation of the application design.

We considered the following secure design patterns documented in [22] for the evaluation of implementation errors:

1) **Secure Factory** separates the security logic of creating an object from the object's basic functionality.
2) **Secure Chain of Responsibility** decouples the logic that determines user/environment-trust functionality from the requester of the functionality.
3) **Secure Builder Factory** separates the security rules involved in creating a complex object from the basic steps involved in creating the object.

D. SECURE FACTORY IMPLEMENTATION

Figure 3 presents the structure of the Secure Factory pattern. This pattern separates the security logic employed in creating or selecting an object from the object's basic functionality.

We evaluate the errors originating from pattern interpretation by programmers, pattern translation to the application design, and later from the application design to code. Table 1 presents the correspondence of each evaluation parameter with the implementation phases.

1) USE CASE

The use case implemented by programmers concerns the creation of a *User* object (Figure 4) by a *ConcreteSecureFactory*

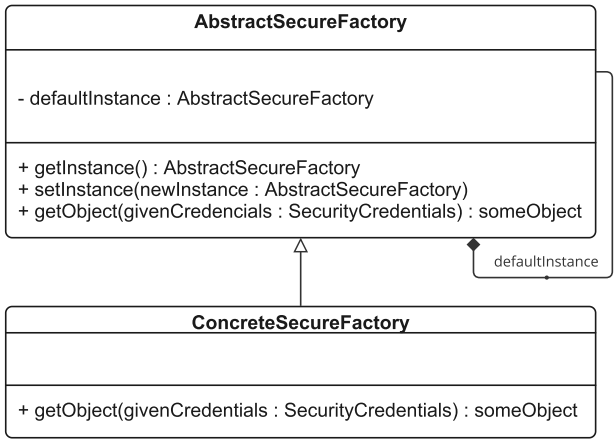

FIGURE 3. Secure Factory design pattern.

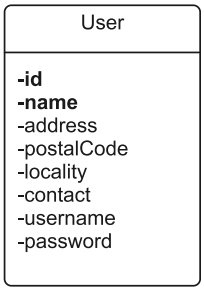

FIGURE 4. User class implemented in the Secure Factory use case. Attributes formatted as bold are accessible by any user.

instance set by the system. When requesting the object, the factory creates the object according to the client credentials provided to the *getObject()* method. The credentials in the use case are JSON Web Tokens (JWT) [23] with the user profile information. If the client owns the user data requested, all attributes presented in Figure 4 will be included in the object populated with the corresponding data. Otherwise, only public data (attributes formatted as bold) are loaded into the object.

2) RESULTS

The results shown in the graph of Figure 5 expose the errors found during each pattern implementation phase. The evaluation of the parameter *pattern well understood* reveals that approximately two-thirds of the programmers could explain the pattern after interpreting and implementing it. The evaluation of the parameter *getObject() returns object aligned with user profile* shows that the logic was correctly implemented in almost all implementations. On the contrary, the other parameters indicate that the pattern's structure was not respected.

E. SECURE CHAIN OF RESPONSIBILITY IMPLEMENTATION

Figure 6 presents the structure of the Secure Chain of Responsibility pattern. This pattern decouples the functionality provided by the interface to the client from the dynamic system-trust-dependent functionality. Hence, the system can

TABLE 1. Mapping between Secure Factory evaluation parameters and the design pattern implementation phases.

	design pattern understandability	design pattern to application design	application design to code
creation of common object interface		X	
setInstance() sets the concrete factory as default	X		X
getObject() receives credentials	X	X	
getInstance() returns the concrete factory default	X		X
getObject() creates object representation according to user profile	X		X
pattern well understood (subjective expert evaluation)	X	X	

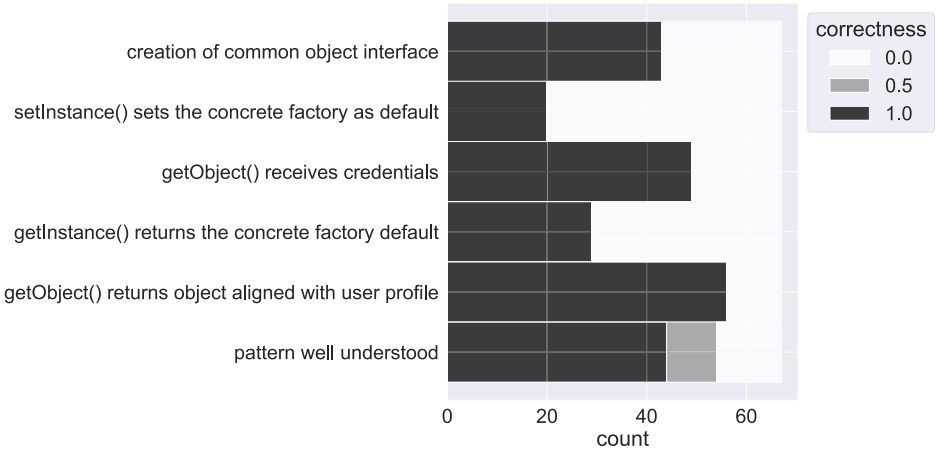

FIGURE 5. Results of the use case implemented using the Secure Factory pattern.

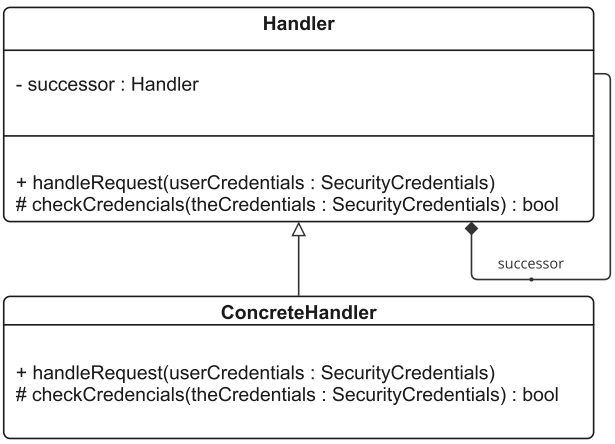

FIGURE 6. Secure Chain of Responsability pattern.

change the functionality according to the credentials provided by the client when it requests the functionality.

1) USE CASE

The use case devised for the Secure Chain of Responsibility patterns employs a gateway to an interface implementation of CRUD (create, read, update, and delete) operations and an operation to change one's user password (Figure 7).

According to the pattern, each operation can have a chain of implementations. When the operation called does not meet the requirements to be executed, the execution control is transferred to the following chain implementation recursively until finding an implementation with valid execution requirements. In the use case, there are security requirements to execute each operation implementation, as such:

- *addUser()* and *removeUser()* are accessible only to *admin* users.
- *searchUser()* is accessible only to *admin* and *admin-limited* users. When executed by *admin-limited* users, a log entry should be created with the *timestamp* and the method parameters.
- *changePassword()* is accessible to the owner of the user data associated with the *user* role.

Any role can execute the implementation at the end of the chain, which triggers an exception. Any user role after being transferred between previous implementations – due to lack of permission to execute them – reaches this implementation. Thus, all operations have at least two implementations. As an example, the *searchUser()* operation has the following implementations (Figure 8):

1) search without logging for *admin* users.
2) search with logging for *admin-limited* users.
3) trigger exception for all other users.

Table 2 presents the correspondence of each evaluation parameter with the pattern interpretation by programmers, pattern translation to the application design, and from the application design to code phases where errors can originate.

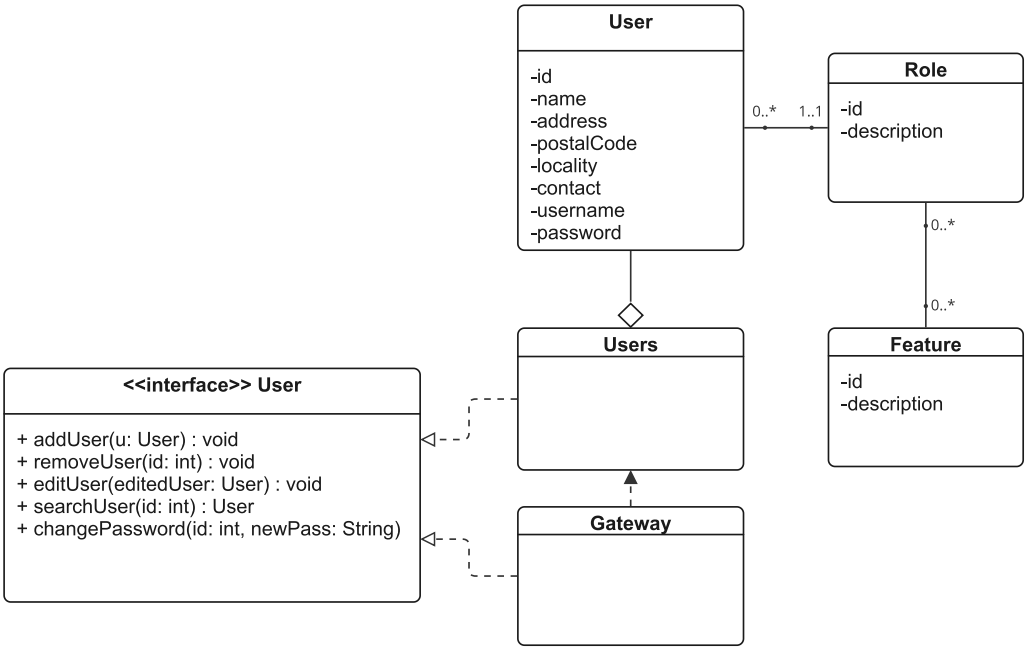

FIGURE 7. Users functionality implemented in the Secure Chain of Responsibility use case.

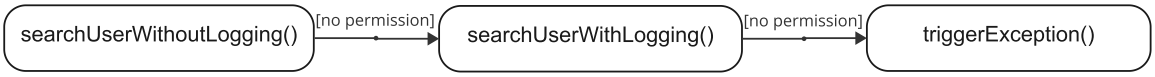

FIGURE 8. Example of transference between implementations due to lack of permissions.

TABLE 2. Mapping between Secure Chain of Responsibility evaluation parameters and the design pattern implementation phases.

	design pattern understandability	design pattern to application design	application design to code
handler/concreteHandler classes are implemented	X	X	
handlers receive security credentials	X	X	
at least 2 implementations of each operation		X	X
handlers reference their successor	X	X	
execution is correctly transferred to successors	X		X
pattern well understood (subjective expert evaluation)	X	X	

2) RESULTS

The evaluation results of the Secure Chain of Responsibility Pattern (Figure 12) are better than those obtained for the previous design pattern. Being the use case complexity higher than that chosen for the Secure Factory – in terms of the number of structural elements involved in the use case – the better results can be explained by the experience obtained from implementing the previous pattern. However, some programmers still do not fully understand the pattern and make mistakes when mapping the pattern design to the application design.

F. SECURE BUILDER FACTORY IMPLEMENTATION

Figure 10 presents the structure of the Secure Builder pattern. This pattern separates the steps involved in creating a complex object that uses several simpler objects from the security rules involved in that creation. The complex object is built by a Builder object selected by the environment according to the security credentials given by the requester client.

1) USE CASE

The use case of the Builder pattern (Figure 11) consists of a gateway to creating complex objects with a security role and all hierarchically ascendant roles (e.g., *admin* is an ascendant of *admin-limited*). Each role object also references its authorized features.

Building the role hierarchy with the corresponding authorized features requires several steps. These steps must conform to the following security requirements:

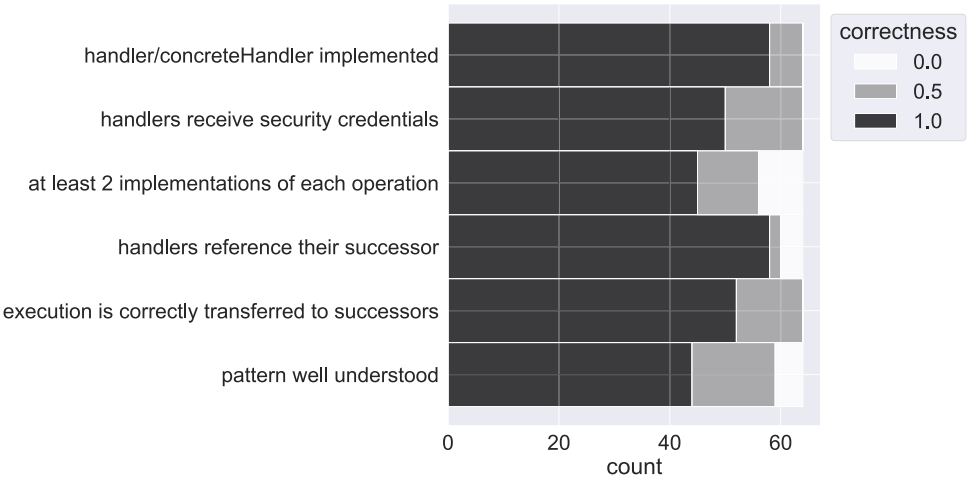

FIGURE 9. Results of the use case implemented using the Secure Chain of Responsibility pattern.

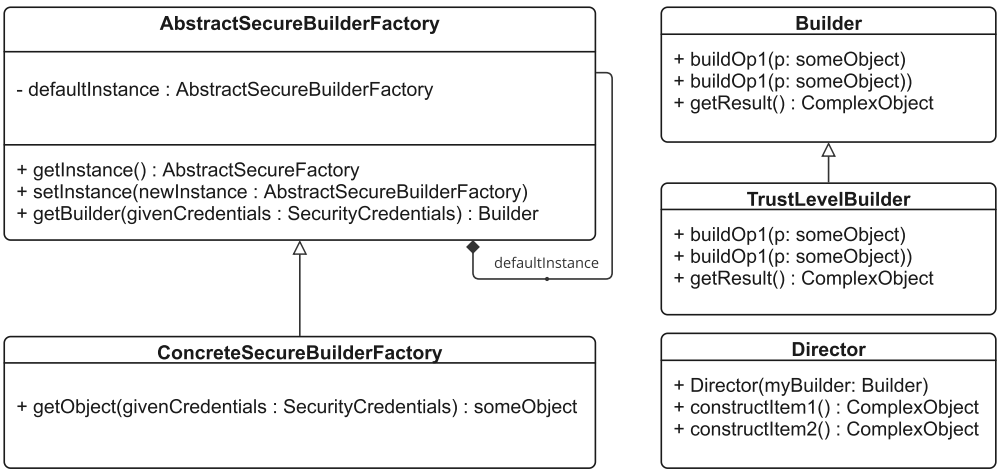

FIGURE 10. Secure Builder Factory pattern.

- only *admin* users can have access to the *createRole()* method.
- the services list administrated by each *admin* user should be part of credentials passed to the invoked method.
- the user must obtain only the roles and features of the services they administrate.

The security requirements force authorization verification to execute the method implemented by the gateway and filter only the data the user has permission to access. Table 3 presents the correspondence of each evaluation parameter with the pattern interpretation by programmers, pattern translation to the application design, and from the application design to code phases where errors can originate.

2) RESULTS

The Secure Builder Pattern exhibited the worst pattern understandability results (Figure 12). Even though there is experience acquired from implementing the previous patterns, some programmers still struggle to fully understand and correctly apply the Secure Builder Pattern. The higher structural complexity of this pattern when compared with the previous ones – the interaction workflow includes the Builder, the Factory, and the Director – explains the results.

G. ANALYSIS OF RESULTS

Evaluation of the secure patterns implemented by junior programmers has shown that despite the pattern logic being correct in almost all implementations, the pattern design does not conform with several. Results of evaluation parameters pinpoint the mapping between design patterns and the application design as the primary contributor to design errors. That means that despite secure design patterns representing a fundamental approach to developing secure applications, their mapping to application design requires validation mechanisms to ensure that the application benefits from the security features provided by the pattern. That validation mechanism will prove the answers to research question RQ2.

TABLE 3. Mapping between Secure Builder evaluation parameters and the design pattern implementation phases.

	design pattern understandability	design pattern to application design	application design to code
pattern classes are correctly implemented	X	X	
getBuilder() receive security credentials	X	X	
there is one concrete builder per service		X	X
getResult() returns the combined results	X	X	
execution workflow is correctly implemented	X	X	X
pattern well understood (subjective expert evaluation)	X	X	

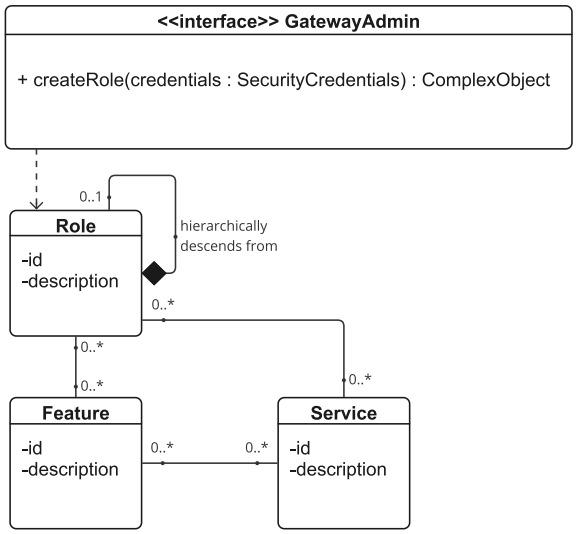

FIGURE 11. Secure Builder use case.

IV. REUSABLE TEST CASE IMPLEMENTATIONS

By incorporating security design patterns into the software development process, developers can ensure that security is a key consideration throughout the software development lifecycle and that security is not an afterthought addressed only after the system has been deployed. However, as exposed by the results of their implementation by several programmers in Section III, there are errors resulting from the interpretation of a design pattern and its mapping to the application design that may compromise the application security requirements. Manual validation of design patterns is a common approach to validating design pattern implementations, but it can be expensive and error-prone.

A. AUTOMATED TESTING

Automated testing plays a critical role in ensuring the quality and reliability of software applications. There are several benefits to applying automated testing to validate security requirements. First, it can help to identify security issues early in the development process, when they are typically less expensive to fix. Second, it can help to ensure that security requirements are met. Finally, it can help improve the software system's overall quality by identifying and addressing security issues that could impact its performance, reliability, and maintainability.

Validation of design pattern implementations by resorting to automatic testing is still error-prone. When the programmer or the tester writes tests to validate the application's design structurally, they can create test cases biased towards wrong pattern interpretations and flawed pattern mapping to the application design.

Considering pattern specification as the ground truth to ensure that the security requirements held by the pattern are implemented, the testing process demands an equivalent ground truth that guarantees correct pattern implementation.

Reusable test cases with validated implementations (e.g., through peer validation and throughout code inspections in different projects) can give the ground truth to guarantee the security requirements at the implementation phase.

B. REUSABLE TEST CASES

The space of the application design corresponding to each pattern specification is ample. Thus, applying reusable test cases to the application design requires a mechanism to identify the structural pattern elements incorporated in the application. Model-based testing can ensure automated testing based on a model of the mapping, but models are expensive to build and maintain. Plus, they demand an additional validation effort to avoid model-level security flaws.

Our approach is based on test cases that validate the presence of structural pattern elements marked with reusable Java annotations [24]. The reusable test cases are written in Java using the JUnit framework [25]. The Reflections library [26] provides information on classes, methods, and fields marked with specific annotations to be used by reusable test cases.

Listing 1 presents a reusable test case to validate the existence of the Secure Factory pattern classes. It also presents the abstract and concrete classes of the pattern implementation annotated with the *@AbstractFactory* and *@ConcreteFactory* annotations, respectively. The test case asserts the existence of one class marked with each of these annotations and the inheritance relationship between these two classes.

All patterns have reusable test implementations that use annotations as anchors to identify structural pattern elements.

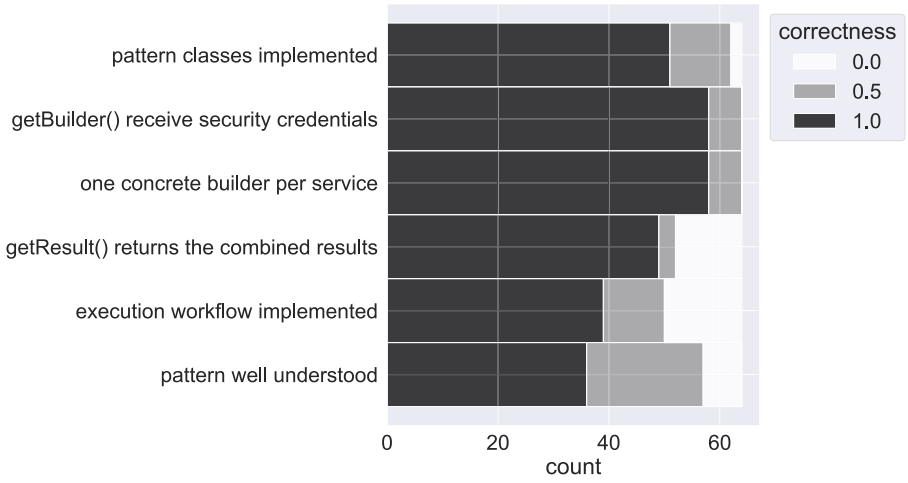

FIGURE 12. Results of the use case implemented using the Secure Builder pattern.

```
// Annotated pattern implementation

@AbstractFactory
public class UsersAbstractFactory {
   // ...
}

@ConcreteFactory
public class UsersConcreteFactory {
   //...
}

/* Reusable test case to validate the inheritance
   relationship between the Abstract Factory and
   Concrete Factory classes in the Secure Factory
   pattern. */

@BeforeAll
public static void setup(){
   reflections =
      new Reflections("com.securityfactory");
   annotatedAbstract = reflections
      .getTypesAnnotatedWith(AbstractFactory.class);
   annotatedConcrete = reflections
      .getTypesAnnotatedWith(ConcreteFactory.class);
}

@Test
public void testAbstractConcreteInheritance(){
   assertTrue(annotatedConcrete.size()>0);
   for(Class c: annotatedConcrete){
      assertEquals(c.getGenericSuperclass()
         .getTypeName(),
      annotatedAbstract.getClass()
         .getTypeName());
    }
}
```

Listing 1. Pattern implementation and reusable test case.

Table 4 presents the complete list of annotations and respective test case validations for each secure pattern. Structural validations of the pattern implementation are broken down into (1) inheritance relationships, (2) placement of methods and fields described in the pattern, and (3) correct field types and correct method parameter and return types. For example, for the Secure Factory pattern, the *@ConcreteFactory* class should be a subclass of an *@AbstractFactory* class. The method annotated with *@GetObjectMethod* should be placed in the *@ConcreteFactory* and *@AbstractFactory* classes. It also should have a parameter of type *@Credential*.

C. EXPERIMENTAL EVALUATION AND DISCUSSION

All patterns' structural rules and constraints have been implemented as reusable tests. We reassign pattern implementations to the programmers again by providing the tests and asking them to incorporate annotations in their use case implementations. We did not explain what was wrong with their previous secure pattern implementations to keep the assignment bias-free toward new knowledge.

As expected, all use case implementations were free of structural errors. Plus, the annotations have driven the pattern inclusion into the use case. They are regarded not only as additional pattern documentation but also as a framework to structure use case implementations designed according to the pattern specification. Also, reusable tests work as regression tests that reinforce structural correctness after software changes resulting from their maintenance.

Automated reusable tests are closely related to refactoring in software development. Refactoring involves improving the structure and design of existing code without changing its external behavior. It aims to enhance code quality, readability, and maintainability. When refactoring, developers often need to make changes to the code, which can introduce new bugs or unintended behavior. Automated reusable tests play a crucial role in ensuring that the refactored code still functions correctly. They help verify the code's behavior, act as a safety net, detect regressions, and ensure the stability of the refactored code. By relying on automated tests, developers can confidently refactor a bad design into a good one, knowing that they have a mechanism to verify their changes' correctness.

TABLE 4. Structural validation performed by the reusable test cases for each secure design pattern. It shows, for each pattern, the (1) classes that are associated by inheritance, (2) the classes where the methods and fields show to be placed, and (3) field types, method argument types, and method return types.

		inheritance	correct placement of methods/fields	correct field and argument/return types of constructors and methods
Secure Factory	@AbstractFactory	X	$X^{1,2}$	$X^{1,2,3}$
	@ConcreteFactory	X	X^{1}	
	@GetObjectMethod		X^{1}	X^{*}
	@DefaultInstanceField		X^{2}	X^{1}
	@GetInstanceMethod		X^{1}	X^{2}
	@SetInstanceMethod		X^{1}	X^{3}
Secure Chain of Responsibility	@AbstractHandler	X	$X^{1,2}$	X
	@ConcreteHandler	X	X^{2}	
	@SucessorField		X^{1}	X
	@HandleRequestMethod		X^{2}	X^{*}
Secure Builder	@AbstractBuilderFactory	X	$X^{1,2}$	$X^{1,2,3}$
	@ConcreteBuilderFactory	X	X^{2}	
	@DefaultInstanceField		X^{1}	X^{1}
	@GetInstanceMethod		X^{1}	X^{2}
	@SetInstanceMethod		X^{1}	X^{3}
	@GetBuilderMethod		X^{2}	$X^{*,4}$
	@AbstractBuilder	X	X^{3}	$X^{4,5}$
	@ConcreteBuilder	X	X^{3}	
	@GetResultMethod		X^{3}	
	@Director		X^{4}	X^{5} (constructor)
	@ConstructItemMethod		X^{4}	
Generic	@Credential			X^{*}

V. CONCLUSION

Security by design is a software and system approach that prioritizes security and privacy. It aims to build inherently secure software and systems resistant to cyber-attacks rather than relying on security patches or retroactive fixes. Secure design patterns represent a promising tool to implement security by design by uniting the security properties with other software quality attributes, such as flexibility, maintainability, and scalability.

Despite the advantages of using design patterns to build safe applications, their implementation was proven to be error-prone. A study of implementing three secure design patterns by 70 programmers revealed errors resulting from pattern misinterpretation and difficulties mapping its design specification to the application domain. We propose using reusable test case implementations and annotations to identify patterns' structural elements in the application code. The annotations work as a framework to (1) reinforce the correct implementation of pattern elements, (2) extend the pattern documentation, and (3) promote easy software maintainability by validating the pattern structure between versions of the application. However, annotations should be supported by the programming language to create structural anchors for automated testing (Java, C#, Python, Ruby and VB.NET are examples of languages that support anchors). Also, annotations should be expressive enough to allow annotations of important structural elements (e.g., classes and methods).

Secure design patterns encapsulate reusable design solutions for recurrent problems. We proved that it is possible to implement reusable test cases for validating their correct application to specific use cases. However, errors not originating at the structural level can still occur since our test cases do not cover them. We aim to work on solutions to avoid these error types in future work.

REFERENCES

[1] *7 Top Trends in Cybersecurity for 2022*, Gartner, USA, 2022.
[2] *6th Annual Hacker-Powered Security Report*, Hackerone, USA, 2022.
[3] D. Gonzalez, F. Alhenaki, and M. Mirakhorli, "Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Mar. 2019, pp. 31–40.
[4] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proc. 10th Int. Conf. Quality Softw.*, Jul. 2010, pp. 23–31.
[5] S. Arzt, "Security and quality: Two sides of the same coin?" in *Proc. 10th ACM SIGPLAN Int. Workshop State Art Program Anal.*, Jun. 2021, pp. 7–12.
[6] H. Krasner, "The cost of poor software quality in the US: A 2020 report," in *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021, pp. 25–26.
[7] K. Tuma, L. Sion, R. Scandariato, and K. Yskout, "Automating the early detection of security design flaws," in *Proc. 23rd ACM/IEEE Int. Conf. Model Driven Eng. Lang. Syst.*, Oct. 2020, pp. 332–342.
[8] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 662–671.
[9] M. Yoshizawa, T. Kobashi, H. Washizaki, Y. Fukazawa, T. Okubo, H. Kaiya, and N. Yoshioka, "Verifying implementation of security design patterns using a test template," in *Proc. 9th Int. Conf. Availability, Rel. Secur.*, Sep. 2014, pp. 178–183.
[10] A. Panjiyar and D. Sadhya, "Defending against code injection attacks using secure design pattern," in *Proc. 29th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2022, pp. 570–571.

[11] I. Ali, M. Asif, M. Shahbaz, A. Khalid, M. Rehman, and A. Guergachi, "Text categorization approach for secure design pattern selection using software requirement specification," *IEEE Access*, vol. 6, pp. 73928–73939, 2018.
[12] O. Ur-Rehman and N. Zivic, "Secure design patterns for security in smart metering systems," in *Proc. IEEE Eur. Model. Symp. (EMS)*, Oct. 2015, pp. 278–283.
[13] C. A. Cunha and J. L. Sobral, "An annotation-based framework for parallel computing," in *Proc. 15th EUROMICRO Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Feb. 2007, pp. 113–120.
[14] C. Xi, B. Harbulot, and J. R. Gurd, "Aspect-oriented support for synchronization in parallel computing," in *Proc. 1st Workshop Linking Aspect Technol. Evol.*, Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, Eds. Bonn, Germany, Mar. 2009, pp. 37–41.
[15] C. A. Cunha, J. L. Sobral, and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *Proc. 5th Int. Conf. Aspect-Oriented Softw. Develop.*, Mar. 2006, pp. 134–145.
[16] P. Antonino, S. Duszynski, C. Jung, and M. Rudolph, "Indicator-based architecture-level security evaluation in a service-oriented environment," in *Proc. 4th Eur. Conf. Softw. Architecture, Companion Volume*, Aug. 2010, pp. 221–228.
[17] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *IEEE Softw.*, vol. 13, no. 6, pp. 47–55, Nov. 1996.
[18] T. Weilkiens, *Systems Engineering With SysML/UML: Modeling, Analysis, Design*. Amsterdam, The Netherlands: Elsevier, 2011.
[19] *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (Square)—System and Software Quality Models*, Standard ISO/IEC 25010, 2011.
[20] D. Rombach, M. Ciolkowski, R. Jeffery, O. Laitenberger, F. McGarry, and F. Shull, "Impact of research on practice in the field of inspections, reviews and walkthroughs: Learning from successful industrial uses," *ACM SIGSOFT Softw. Eng. Notes*, vol. 33, no. 6, pp. 26–35, Oct. 2008.
[21] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.
[22] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, "Secure design patterns," Softw. Eng. Inst., Carnegie Mellon, Univ., Pittsburgh, PA, USA Tech. Rep. CMU/SEI-2009-TR-010, 2009.
[23] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT) Internet-Draft Draft-IETF-OAUTH-JSON-Web-Token-25, Internet Engineering Task Force*, IETF, OAuth Working Group, USA Report no. draft-ietf-oauth-json-web-token-25, 2014.
[24] M. Ernst and D. Coward, "Annotations on Java types," Oracle, Austin, TX, USA, Tech. Rep. JSR 308, 2007.
[25] (Apr. 2023). *Junit 5*. [Online]. Available: https://junit.org/junit5/
[26] (Apr. 2023). *Reflections Library*. [Online]. Available: https://www.javadoc.io/doc/org.reflections/reflections/latest/index.html

CARLOS CUNHA received the Ph.D. degree in informatics and in dependable systems from the University of Coimbra and the Master of Science degree in informatics and in software engineering from the University of Minho. He is currently an Associate Professor with the Polytechnic University of Viseu, Portugal. His research interests include healthcare systems, distributed systems, and software engineering.

NUNO POMBO (Senior Member, IEEE) is currently an Assistant Professor with the University of Beira Interior (UBI), Covilhã, Portugal. His current research interests include information systems (focusing on decision support systems), data fusion, software, software engineering, and software engineering education. He is a member of the BSAFE Laboratory, UBI, where he is also with the IT Network Applications and Services Group, Instituto de Telecomunicações.