Securing Gang of Four Design Patterns

ABBAS JAVAN JAFARI, Ferdowsi University of Mashhad
ABBAS RASOOLZADEGAN, Ferdowsi University of Mashhad

Software design patterns are a means to specify common solutions to reoccurring design problems. Similarly, security design patterns provide a means to encapsulate common security solutions and mechanisms which are applicable at the design phase of the software development lifecycle. Security patterns have received considerable attention from the security community to introduce concepts such as authentication, access control and encryption to software design. Unfortunately, many well-known and commonly-used design patterns have been documented in the absence of security requirements. Using these patterns in the software development lifecycle can introduce new vulnerabilities into the system. Therefore, it is crucial that well-established design patterns such as GoF patterns also evolve to accommodate new security requirements. In this paper, we discuss the current state of security regarding GoF design patterns. We then propose a general methodology for securing design patterns. The proposed methodology is employed to secure the Mediator and Singleton patterns by enhancing their basic GoF counterparts in terms of fundamental security goals such as availability, integrity and confidentiality. Both patterns will undergo a vulnerability analysis phase to guide the extension process. The new versions of the Mediator and Singleton will be analyzed and validated in terms of both security capabilities and design flexibility.

Categories and Subject Descriptors: **• Security and privacy~Software security engineering • Software and its engineering~Design patterns**

General Terms: Design

Additional Key Words and Phrases: Security Patterns, Gang of Four Design Patterns, Secure Mediator, Secure Singleton.

ACM Reference Format:

Jafari, A. J. and Rasoolzadegan, A. 2016. Securing Gang of Four Design Patterns. Proceedings of The 23rd Conference on Pattern Languages of Programs (PLoP 2016)

1. INTRODUCTION

There have been many contributions from both the Software Engineering and Information Security community in introducing security solutions and best practices to the software development lifecycle. Security design patterns provide a means to encapsulate common solutions to reoccurring security problems. They are a means to encapsulate and distribute security knowledge [Hafiz et al. 2007]. A great number of security patterns have already been proposed to tackle security issues such as authentication [Brown et al. 1999], access control [Priebe et al. 2004], encryption [Schumacher et al. 2013], privacy [Hafiz 2006], firewall [Villarreal et al. 2013] and accounting [Fernandez et al. 2011]. One of the most complete catalogs of security design patterns which also describes a secure development methodology can be found in [Fernandez 2013]. However, the existing software design patterns have received considerably less attention regarding security vulnerabilities and defense mechanisms. Many prominent design patterns such as the Mediator and Singleton have not yet been considered for security analysis. Some patterns such as the Abstract Factory and Strategy have been considered for security enhancement [Dougherty et al. 2009], but these enhancements only change the basic pattern functionality based on user credentials to secure the pattern. The approach taken in this work does not analyze the potential vulnerabilities faced by each design pattern or what countermeasures should be in place to prevent them. This is unfortunate, as many of these design patterns are well-established and are being used by developers to tackle software design problems. Among all available design patterns, the prominence of GoF patterns is evident. Apart from their importance in tackling software design problems, GoF patterns are also the basis for many current security patterns [Blakley and Heath 2004; Schumacher, et al. 2013]. It is therefore beneficial to analyze the potential vulnerabilities of GoF patterns and provide security enhancements where it is needed. For this reason, we will first review the current state of GoF patterns with regard to security. We will then introduce a general methodology for securing current design patterns. We will use the same methodology to present two new extensions to secure the Mediator and Singleton patterns. This paper emphasizes on secure design patterns rather than security patterns. Secure design patterns aim to eliminate the accidental insertion of vulnerabilities due to design pattern usage [Dougherty, et al. 2009]. They differ from security design patterns in that they don't focus on implementing specific security concepts such as access control or authentication. Thus, secure design patterns and security design patterns are differentiated based on intent.

The approach in this paper takes a Gang of Four design pattern as an input and after following the proposed methodology, it outputs a secure extension of that pattern as a result. Augmenting a design pattern with other

quality factors is carried out in other research works such as in [Buckley et al. 2011], where security patterns are augmented with a fault-tolerance mechanism to ensure reliable security functionality.

Section II of the paper discusses previous works that focus on securing GoF design patterns and various security patterns that utilize GoF patterns. Section III introduces the general methodology for analyzing and securing design patterns. This section also provides security extensions on the Mediator and Singleton patterns using the same methodology. The verification and validation of the proposed patterns are discussed at the end of Section III. We conclude our work in Section IV and discuss potential future works in Section V.

2. RELATED WORKS

There have been various works which focus on securing GoF design patterns. Fernandez and Ortega-Arjona have utilized access control and authentication mechanisms to secure the Adapter pattern [Fernandez and Ortega-Arjona 2009]. Gondi has proposed a secure Observer pattern for distributed systems by complementing the base pattern with an authentication mechanism and encrypted communications [Gondi 2010]. Dougherty et.al. have introduced extensions to the Builder, Strategy, Abstract Factory, Chain of Responsibility, State, and Visitor patterns. The extensions aim to furnish these patterns for security-critical environments by separating the security logic for creating and managing the objects, from the basic functionality of the design pattern. For example, the Secure Strategy pattern provides a means to easily select and modify the appropriate strategy object to perform a task based on the security credentials of the current user. This is accomplished by combining authentication mechanisms with the base Strategy pattern to handle requests based on the user's trust level [Dougherty, et al. 2009]. However, The pattern extensions in [Dougherty, et al. 2009] do not cover the potential vulnerabilities that can be introduced by using each pattern and the solutions to tackle those vulnerabilities.

Gang of Four design patterns are also the basis for many current security patterns. The Replicated System and Error Detection/Correction [Blakley and Heath 2004] security patterns both utilize the GoF Proxy pattern [Gamma et al. 1995]. The Protected System [Blakley and Heath 2004] and Single Access Point [Schumacher, et al. 2013] patterns employ the concept of the GoF Façade pattern [Gamma, et al. 1995] to implement the system guard. The Account Lockout pattern [Kienzle et al. 2002] uses the concept of the GoF Mediator pattern [Gamma, et al. 1995] to check and manage different components of the lockout mechanism. This suggests that a security flaw in the employed design patterns can compromise the security of the resulting security patterns.

3. SECURING DESIGN PATTERNS

This section presents a general methodology for securing software design patterns. Fig. 1 displays the steps needed to achieve a secure design pattern using a UML activity diagram. Once we have chosen a design pattern for augmentation, we begin the vulnerability analysis. The specific approach for this step is a decision we leave up to the analyst. We have chosen to analyze the potential threats to the three main security objectives (confidentiality, integrity, availability). The threats are categorized based on the STRIDE threat assessment model [Swiderski and Snyder 2004]. We have also mentioned specific vulnerabilities and attacks where applicable (e.g. DDoS attack). After analyzing potential vulnerabilities, we decide which security objectives to address based on the pattern's core structure and intent. It is important to note that each pattern has different objectives. Therefore, the vulnerability analysis is specific to each pattern and will differ from one case to another.

The next step consists of searching for security solutions based on the chosen vulnerabilities and security objectives. Many security solutions have already been proposed in the form of security patterns. This can simplify the augmentation step as security patterns are closer to the software design realm. However, we cannot assume that every desired solution is already presented in the form of a security pattern.

The augmentation step extends the chosen design pattern with the selected security mechanisms. If the solution is chosen from outside the software engineering domain (no security patterns exists for the solution), then the augmentation step requires additional effort to convey the security knowledge to the software design domain. In cases where the desired security pattern exists, augmentation still requires diligent customization before application.

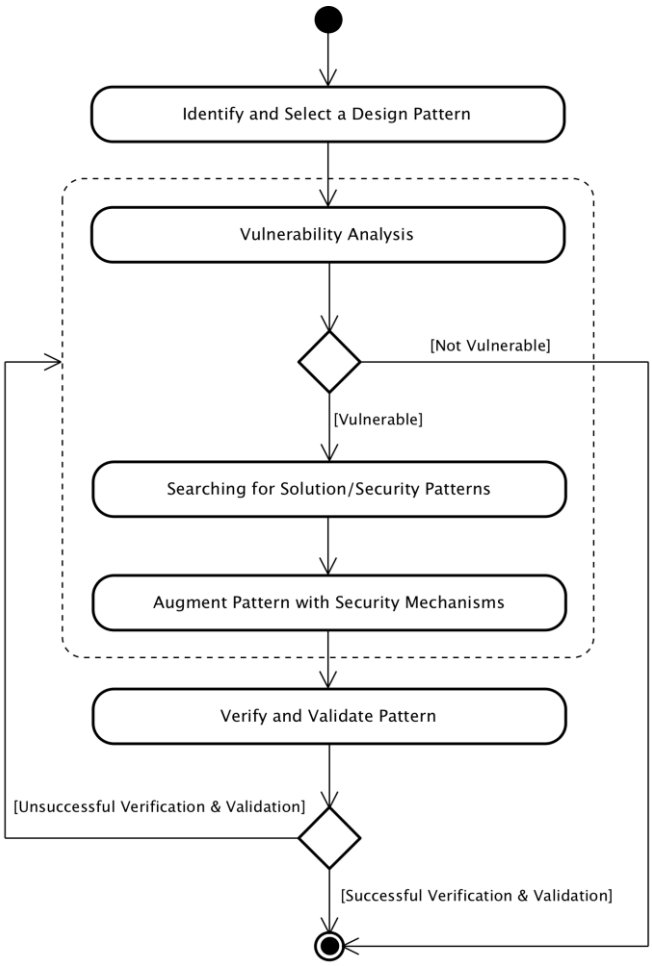

Fig. 1. Methodology for securing design patterns

The final step involves the verification and validation of the newly secured version of the design pattern. The new pattern should satisfy the security objectives chosen in the vulnerability analysis stage. It is also important that the extended pattern provides the same order of flexibility as the base design pattern. In other words, securing a design pattern should not hinder the primary objectives of software design patterns (flexibility and reusability). If the verification and validation fails to satisfy both the security and flexibility criteria, the methodology falls back to one of previous stages (based on the analyst's decision). The verification and validation step can also be carried out by a third-party individual.

3.1 Secure Mediator Pattern

The Mediator pattern aims to define an object that is responsible for managing how other objects interact. This scheme promotes loose coupling and simplifies future modifications. The primary compromise with this approach is the creation of a complex and monolithic object (the Mediator) which can be hard to maintain [Gamma, et al. 1995]. From a security perspective, the availability of this object is crucial to the availability of the subsystem. Therefore, the Mediator becomes an attractive target for attacks.

3.1.1 Vulnerability Analysis

The vulnerabilities of the Mediator pattern (like many other GoF patterns) can be discussed from different viewpoints. The availability of the Mediator can directly affect the availability of the subsystem. In a scenario where the colleagues are externally connected units, the confidentiality of message transmission between the Mediator and its colleagues can also be taken into account. One can also argue that the integrity of the Mediator and its operations is also a relevant criterion for vulnerability assessment. However, adding all possible security mechanisms to minimize vulnerabilities is inefficient. Satisfying each of the Availability, Confidentiality and Integrity objectives requires additional classes, methods, and attributes to be added to the

Mediator pattern. Additionally, the extended security pattern should be suitable for a wide array of use cases. Adding too many security mechanisms narrows the possible applications of the pattern to specific instances (where all security objectives are required). We propose choosing the most important security objectives based on the pattern's intent. The Mediator aims to reduce coupling by centralizing the previously distributed behavior into one (or more) manager object(s) [Gamma, et al. 1995]. This centralization implies the increased need for object availability. If the Mediator fails, all dependent colleagues will lose correct functionality. Therefore, the availability objective is more crucial for the Mediator pattern. One of the major threats to system availability comes from Denial of Service attacks. The attacker can overload the Mediator with multiple requests or send a single malformed request, causing the system to crash. In both cases, DoS attacks are an important external cause for failure. The current state of the Mediator pattern is vulnerable to Denial of Service attacks from the STRIDE threat assessment model [Swiderski and Snyder 2004]. To tackle this threat, we have hardened the pattern structure with a fault-tolerance mechanism. It's worth mentioning that adding a fault-tolerance mechanism focuses on preventing the DoS attack from causing system failure, rather than preventing the attack itself.

We propose the Secure Mediator pattern which extends the base Mediator pattern with the needed security mechanisms. The primary aim of this pattern is to satisfy the availability aspect of security in the Mediator structure. The Secure Mediator pattern employs concepts from the Standby [Blakley and Heath 2004] and Memento [Gamma, et al. 1995] patterns. The Standby security pattern introduces a structure where the service capabilities of object X can be replaced by object Y. The Memento pattern (also used in the Standby pattern) ensures that object Y can continue the same operations of object X after failure.

3.1.2 Pattern Structure

The static structure of the Secure Mediator pattern is presented in the class diagram of Fig. 2. The shaded classes display the structure of the GoF Mediator pattern. Both the Primary and Secondary Mediators are inherited from the same concrete Mediator class. This is to ensure that both objects can handle the same tasks. The Recovery Proxy receives the requests from the colleagues and delegates them to the Primary Mediator. This separation is necessary so that in case of failure in the Primary Mediator, the Recovery Proxy can successfully replace the compromised Mediator with the Secondary Mediator. The Memento pattern is used to retain the state of the currently active Mediator. If the Primary Mediator fails, the Memento will help in bringing the Secondary Mediator up to date, so it can continue the previous set of tasks without interruption.

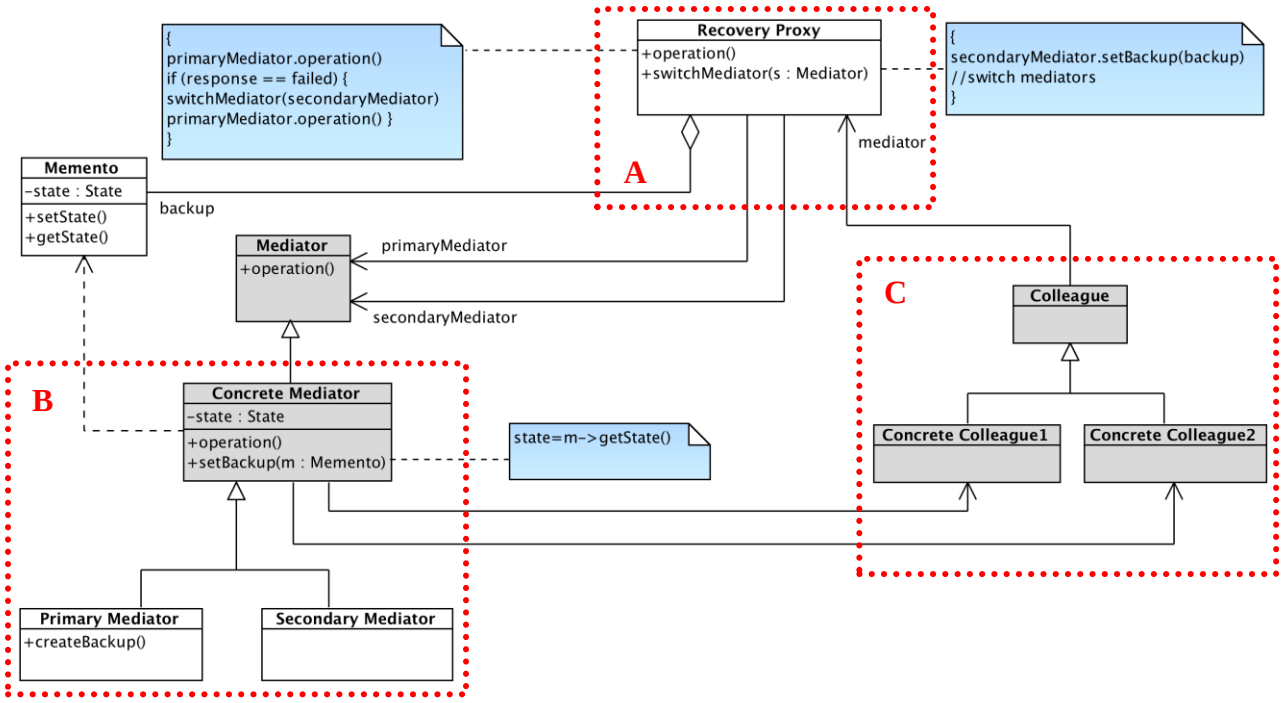

Fig. 2. Secure Mediator Class diagram

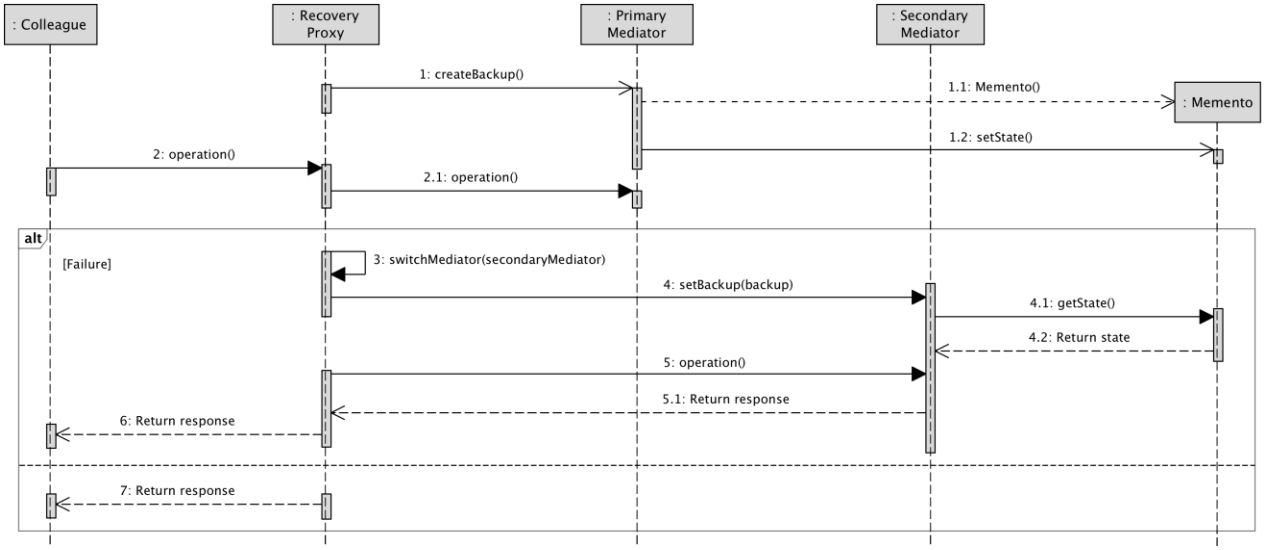

Fig. 3. Secure Mediator Sequence Diagram

The proposed pattern structure provides a means to increase the availability of the Mediator object. However, this structure is based on the premise that various units (displayed using the red dotted lines in Fig. 2) are deployed on different machines. If units A and B are deployed on the same machine, failure of the Primary Mediator (due to physical failure or high work-load) will likely cause a failure in the Recovery Proxy; meaning that the switching process can no longer be accomplished. Additionally, if units B and C reside on the same machine, failure in the Mediator will likely cause a failure in the Colleagues, thus there are no longer any requests for the Mediator to handle. The separation of the structure can further continue to separate the Primary and Secondary Mediators into two separate units, but this is not necessary for software-level failures.

The dynamic structure of the Secure Mediator pattern is displayed using the sequence diagram in Fig. 3. The Recovery Proxy delegates the requests and messages between the Colleagues and the Primary Mediator. In the case that the Primary Mediator fails, the Recovery Proxy configures the Secondary Mediator with the current state of the Memento and forwards the request to the Secondary Mediator, which will now act as the Primary Mediator. This is essentially a role reversal between the Primary and Secondary Mediator objects. From this point onwards, normal system operation is resumed.

3.1.3 Consequences

Using the proposed pattern extension produces the following consequences:

- The security of the Mediator pattern has been improved through the use of a fault-tolerance mechanism (further explained in the Verification and Validation section)
- Due to the added classes, the cost for implementing the pattern has increased (further explained in the Implementation Discussion)
- The communication complexity between the Mediator and colleagues has increased

3.1.4 Implementation Discussion

By observing the current variant of the Secure Mediator pattern, and comparing it with the basic GoF pattern, it is clear that by adding a fault-tolerance mechanism, we have introduced significant overhead to the pattern. Each Secure Mediator will have an increased number of classes, attributes, and methods when compared to the initial GoF pattern. Furthermore, the extended structure increases the communication complexity between the Mediator and its Colleagues. Thus, in a system utilizing multiple Mediators, we must be very selective in choosing which Mediators should be fault-tolerant and which ones should not. The selection process will depend on the type of application (critical vs trivial) and the duties handled by the Mediator (critical vs trivial).

Additionally, the frequency of backing up the Primary Mediator's state in the Memento can greatly harm performance, especially if the backup is stored in a separate unit from the Mediator. On the other hand, too infrequent backups to the Memento will cause a greater amount of state information to be lost in the case of failure.

The current variant of the Secure Mediator pattern only allows the Primary Mediator to create the Memento object. This is because the Secondary Mediator should only receive and assign the Memento without creating a new object. In cases where we should only instantiate one Memento, but not necessarily through the Primary Mediator, it is preferable to combine the Singleton and Memento patterns to create a Singleton Memento.

The Memento provides a wide interface (Mediator) and narrow interface (Recovery Proxy). Ideally, the Recovery Proxy only maintains a pointer to the Memento and only the Mediators should have access to the *set* and *get* operations in the Memento [Gamma, et al. 1995]. In a C/C++ implementation, it is possible to declare the Memento's methods as private and define a friend relationship between the Mediator and the Memento. Other languages may provide similar solutions.

3.2 Secure Singleton Pattern

The Singleton pattern aims to ensure that only one instance of a class is instantiated, and provides a global point of access to that object. This permits a more controlled access to the desired object [Gamma, et al. 1995]. The Singleton pattern is commonly used when one instance of a class is used by multiple components of the system. For instance, different components of a system will usually employ a singleton file system or a singleton window manager. This creates a potential for system vulnerability, as any alteration of the Singleton object (before or after instantiation) can have cascading effects on the whole system.

3.2.1 Vulnerability Analysis

The vulnerability analysis for the Singleton pattern follows a similar approach to the vulnerability analysis of the Mediator pattern. The Singleton pattern provides a global point of access to a single instance of a class. This suggests that other components of the system are dependent on the correct instantiation and functionality of the Singleton object. Similar to the Mediator pattern, all three security objectives are beneficial to the Singleton. However, based on the pattern's intent, we have chosen the confidentiality and integrity objectives. Due to its global nature, many clients will likely request the Singleton instance. It is therefore necessary to ensure that only authorized clients can acquire (or create) this instance (through some form of access control). From the opposite standpoint, the clients depend on the correct state and version of the Singleton to maintain healthy functionality. Therefore, it is also necessary for the clients to ensure they have received a legitimate and untampered Singleton instance. This implies the need for an authentication mechanism. The lack of an authentication and access control mechanism means that the current state of the Singleton pattern is vulnerable to Spoofing, Data Tampering, Information Disclosure and Escalation of Privilege attacks from the STRIDE threat assessment model [Swiderski and Snyder 2004]. Adding access control and authentication to the pattern can also create a means to safeguard the pattern from Repudiation attacks, but this requires additional logging mechanisms, which we have chosen to exclude due to cost and maintaining pattern generality (similar to the discussion on the Mediator pattern).

We propose the Secure Singleton pattern which extends the base pattern by adding access control and authentication mechanisms. This enhances the basic Singleton pattern in terms of both confidentiality and integrity. The Secure Singleton pattern employs concepts from the Role-Based Access Control [Priebe, et al. 2004] and Authenticator [Brown, et al. 1999] security patterns. The Role-Based Access Control pattern ensures that only authorized clients can access the Singleton object. This access control is necessary at both the instantiation phase and future recalls of the Singleton instance (other forms of access control mechanisms such as Access Matrix or Multi-Level models can also be used). The Authenticator pattern allows the user to verify that the received instance has not been tampered during the transfer. Thus, the following scheme allows mutual verification between the Singleton object and its clients.

3.2.2 Pattern Structure

The static structure of the Secure Singleton pattern is presented in the class diagram of Fig. 4. The shaded classes show the structure of the GoF Singleton pattern. The proposed structure allows the Singleton to verify the security privileges of the client using role-based access control. On the other end, the client can also verify the received Singleton instance to make sure it has received a valid instance. Role-based access control is accomplished by checking each client's privilege level to make sure it has sufficient permission for the request. Singleton instance authentication is achieved by checking the instance's unique ID against the client's authentication information. Each client has a unique ID and each Singleton also has a unique ID. Fig. 5 depicts the dynamic structure of the Secure Singleton pattern in a sequence diagram.

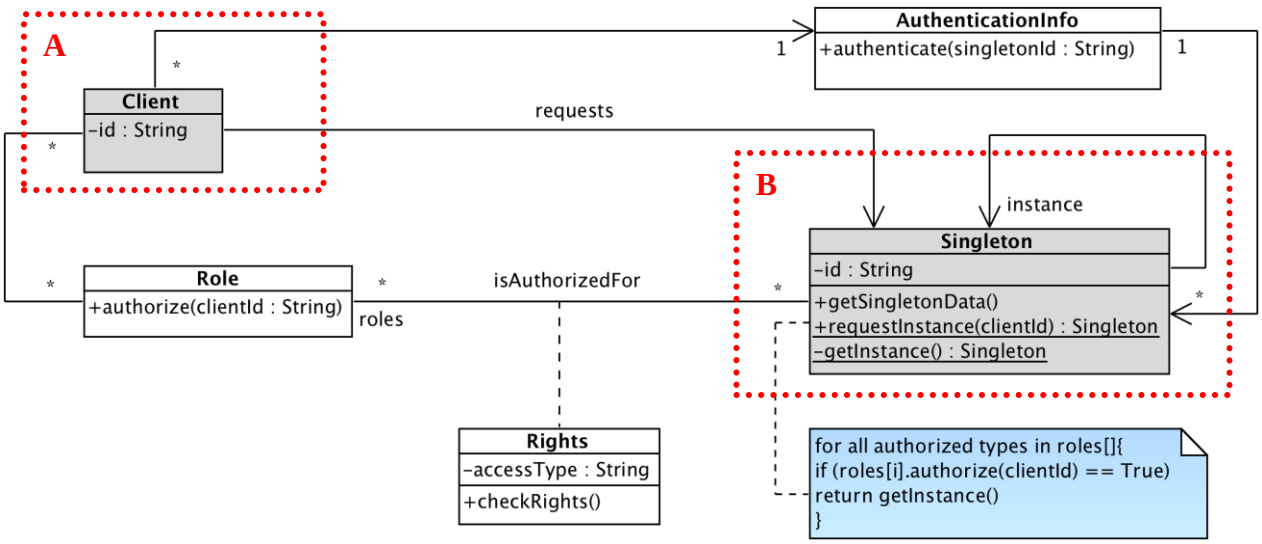

Fig. 4. Secure Singleton Class diagram

Similar to the discussion for the Secure Mediator pattern, the Secure Singleton is comprised of separate units (displayed using the red dotted lines in Fig. 4). The Client (Unit A) can be any external system or externally deployed part of the Singleton's system (Unit B). Otherwise, authorization and authentication would be meaningless because all requests and responses are transmitted and handled internally. The external communication between the two units suggests that unauthorized clients can also request the Singleton instance, and the instance sent to authorized clients can be tampered or spoofed before reaching its destination.

3.2.3 Consequences

Using the proposed pattern extension produces the following consequences:

- The security of the Singleton pattern has been improved through the use of an Access Control (for the clients) and Authentication (for the instances) mechanism (further explained in the Verification and Validation).
- Due to the added classes, the cost for implementing the pattern has increased (further explained in the Implementation Discussion).

3.2.4 Implementation Discussion

The *Authentication Information* must be a trusted entity containing the validation information for every Singleton in the structure. Whether this information is hardcoded in the *Authentication Information* or registered later on depends on specific implementations of this pattern. In any case, the trusted entity must

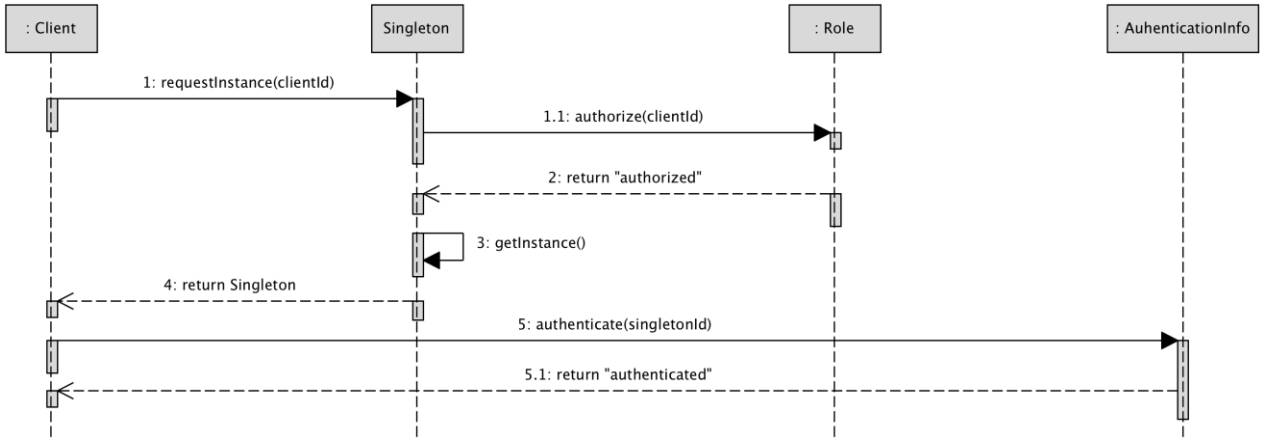

Fig. 5. Secure Singleton Sequence Diagram

reside outside Unit B, otherwise the attacker can tamper with this entity once it has gained access to Unit B.

The current variant of the Secure Singleton does not separate the authority level for instantiating the Singleton, or simply receiving the instance. Authorized roles can instantiate or acquire the Singleton instance. There are plausible situations where different roles must have different degrees of authority when requesting the Singleton instance. It is also preferable to define the amount of access each role has to the Singleton data after receiving the instance.

In the Secure Mediator pattern, we discussed how adding fault-tolerance to every single Mediator in the system can cause significant overhead in performance. The same argument holds true for the Secure Singleton pattern. Both the authorization of the clients and authentication of the Singleton instances requires additional classes, attributes, and methods. The resulting overhead suggests that we must be selective in choosing which Singletons in the system should include the proposed security extensions. Implemented variants can also be more specific in the types of Singleton tasks (critical) which require authorization and the type of tasks (trivial) which do not justify the overhead in performance.

3.3 Verification and Validation

In order to verify the augmentation of security mechanisms to the Mediator and Singleton patterns, we must analyze the threats faced by the GoF pattern, and compare them with the new and secure version of the pattern. The Secure Mediator pattern aims to provide increased availability for the Mediator. For all colleagues that depend on the Mediator for correct functionality (C), the potential damage faced by the system (D) in case of failure of the Mediator can be examined using the following:

$$D = \sum_{colleagues \epsilon C} Colleagues$$

By preventing (or reducing) the failure in the Mediator (by using the fault-tolerance Secure Mediator), the amount of potential damage due to the lack of Mediator availability is reduced. Adding more backup Mediators can further improve availability and reduce the potential damage.

The Secure Singleton aims to increase integrity and confidentiality by only allowing authorized clients to gain access to the Singleton instance. Other clients depend on the correct state and functionality of this instance to maintain correct functionality. For all components that depend on the Singleton instance, the potential risk (R) in case of Singleton manipulation by malicious clients (M) that can access the Singleton can be examined using the following:

$$R = \sum_{clients \epsilon M} Clients$$

This suggests that the potential risk is increased with each malicious client that gains access to the Singleton. By ensuring that only authorized clients can access the Singleton and allowing each client to verify the instance's integrity, we can reduce the potential risk.

The discussion above provides a means to examine the *increase* or *decrease* of potential risk in an application. However, the precise degree of a pattern's effectiveness in reducing security risks depends on the concrete implementation of the pattern. The implementation can be tested to verify that the security enhancements have in fact reduced threat and improved security. Yoshizawa et al. [Yoshizawa et al. 2016] propose a support method for validating the correct implementation of security design patterns in code. Their iterative testing technique ensures that the pattern has been implemented correctly and provides the specified security functionality.

Design patterns are a means to develop more flexible software [Gamma, et al. 1995]. Therefore, enhancing the security of such patterns should not negatively impact the flexibility of the resulting system. We have used the flexibility measurement in [Eden and Mens 2006] to calculate the order of flexibility for the newly secured design patterns. This method measures the order of flexibility for a specific design when faced with a new requirement (evolution step). For each of the proposed patterns, we have defined the cost as the total number of classes subject to modification, for a single instance of the evolution step. Therefore, the order of cost is defined as follows:

$$O(C^1_{Classes}(\varepsilon))$$

In the Mediator pattern, we have defined three types of changes. A change in the overall behavior of the structure, a change in one of the colleagues, and extending relationships between colleagues. The order of cost for each evolution step is displayed in Table. 1. The traditional design in this context refers to a complex structure where each colleague directly connects to others and subsystem behavior is distributed among colleagues with many-to many relationships. In this case, each of the three evolution steps can require modifying a number of other colleagues. The Mediator pattern centralizes behavior; therefore, each evolution step requires modifying only the Mediator. In terms of the Secure Mediator pattern structure, a change in behavior requires modifying both the Concrete Mediator and the Recovery Proxy. This still results to an order of one. A change in a colleague or an extension of relationships among colleagues requires modifying only the Concrete Mediator. Thus, the order of cost for the Secure Mediator remains the same.

Table 1. Flexibility Comparisons for the Mediator Patterns

Evolution Step / Design	Change in behavior ε_1	Change in colleague ε_2	Extending colleague relationships ε_3
Traditional Design	$O(\theta \times \lvert Colleagues \rvert)$ $0 < \theta < 1$	$O(\theta \times \lvert Colleagues \rvert)$ $0 < \theta < 1$	$O(\theta \times \lvert Colleagues \rvert)$ $0 < \theta < 1$
Mediator (GoF)	$O(1)$	$O(1)$	$O(1)$
Secure Mediator	$O(1)$	$O(1)$	$O(1)$

In the Singleton pattern, we have defined one type of change: The number of clients requesting the Singleton instance. The order of cost for this evolution step is displayed in Table. 2. The traditional design in this context refers to a scheme where each client must collaborate with other clients before instantiating a Singleton object to make sure only one instance exists at all times. In this case, adding a new client requires a modification of all other clients, so they can also check this new client prior to instantiating a new Singleton. The Singleton pattern controls the access to the Singleton instance and ensures only one instance of the Singleton exists. Therefore, newly added clients will not cause any modifications in other clients. In terms of the Secure Singleton pattern, a newly added client requires modifying only the Roles class to ensure proper access control. Thus, the order of cost for the Secure Singleton remains the same.

Table 2. Flexibility Comparisons for the Singleton Patterns

Evolution Step / Design	New Client ε
Traditional Design	$O(\lvert Singleton'sClients \rvert)$
Singleton (GoF)	$O(1)$
Secure Singleton	$O(1)$

4. CONCLUSION

Securing well-established design patterns can improve the security of software systems by lowering the likehood of introducing vulnerabilities to the software design process. We have proposed a general methodology for securing design patterns. We have then followed the same methodology to enhance the Mediator and Singleton patterns with regard to security. These extensions are derived from the vulnerability analysis of each pattern based on its intent and structure. We have also validated the new secure patterns to make sure they reduce the security risks and still provide the same order of flexibility as the base patterns.

5. FUTURE WORK

A great number of existing design patterns still lack the much needed security provisions. It is possible to secure other GoF or non-GoF design patterns using a similar approach taken in this paper. Different design patterns follow different goals and target different aspects of a software system. It is therefore plausible to further analyze current design patterns and document the security vulnerabilities faced by each pattern based on their functionality. The general methodology discussed in this work should also be analyzed by further utilizing it in practice to secure other design patterns. It is also self-evident that securing design patterns should not hinder the basic functionality of the pattern. This in turn can create some difficulties as security goals can sometimes conflict with other soft-goals such as flexibility and reusability.

We have discussed the negative and positive consequences (added overhead vs increased security) of using these secure patterns. To accurately measure the performance overhead and security gain trade-offs, we should observe and evaluate the implementation of these patterns in real-world examples. Safety-critical, Mission-critical, and Business-critical systems are a good example where both security and performance is a desired quality. Therefore, applying these patterns to systems such as air traffic control, navigation, or employee administration can give new insights to the effectiveness of these patterns, as well as guidance on how to effectively apply them.

ACKNOWLEDGEMENTS

We would like to thank Prof. Eduardo B. Fernandez for providing useful comments which significantly improved this work. We would also like to thank our colleagues at the Software Quality Lab at Ferdowsi University for their support in preparing this paper.

REFERENCES

BLAKLEY, B. AND HEATH, C. 2004. *Security Design Patterns*. The Open Group Security Forum.

BROWN, F., DIVIETRI, J., DE VILLEGAS, G. AND FERNANDEZ, E.B. 1999. The authenticator pattern. In *Proceedings of Pattern Language of Programs*.

BUCKLEY, I.A., FERNANDEZ, E.B. AND LARRONDO-PETRIE, M.M. 2011. Patterns combining reliability and security. *Procs. of PATTERNS*, 144-150.

DOUGHERTY, C., SAYRE, K., SEACORD, R., SVOBODA, D. AND TOGASHI, K. 2009. Secure design patterns Software Engineering Institute, Carnegie Mellon University.

EDEN, A.H. AND MENS, T. 2006. Measuring software flexibility. *IEE Proceedings - Software 153*, 113-125.

FERNANDEZ, E.B. 2013. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons.

FERNANDEZ, E.B., MUJICA, S. AND VALENZUELA, F. 2011. Two security patterns: least privilege and security logger and auditor. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, Tokyo, Japan2011 ACM, 2524638, 1-6.

FERNANDEZ, E.B. AND ORTEGA-ARJONA, J.L. 2009. Securing the Adapter pattern. In *Procs. of the OOPSLA MiniPLoP*.

GAMMA, E., JOHNSON, R., HELM, R. AND VLISSIDES, J. 1995. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

GONDI, V.B. 2010. Secure chained observer pattern in distributed systems. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, Reno, Nevada, USA2010 ACM, 2493306, 1-9.

HAFIZ, M. 2006. A collection of privacy design patterns. In *Proceedings of the 2006 conference on Pattern languages of programs*, Portland, Oregon, USA2006 ACM, 1415481, 1-13.

HAFIZ, M., ADAMCZYK, P. AND JOHNSON, R.E. 2007. Organizing Security Patterns. *IEEE Software 24*, 52-60.

KIENZLE, D.M., ELDER, M.C., TYREE, D. AND EDWARDS-HEWITT, J. 2002. *Security patterns repository version 1.0*. DARPA, Washington DC.

PRIEBE, T., FERNANDEZ, E.B., MEHLAU, J.I. AND PERNUL, G. 2004. A Pattern System for Access Control. In *Research Directions in Data and Applications Security XVIII: 18th Annual Conference on Data and Applications Security July 25–28, 2004, Sitges, Spain* Springer US, Boston, MA, 235-249.

SCHUMACHER, M., FERNANDEZ, E.B., HYBERTSON, D., BUSCHMANN, F. AND SOMMERLAD, P. 2013. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons.

SWIDERSKI, F. AND SNYDER, W. 2004. *Threat Modeling*. Microsoft Press.

VILLARREAL, I.N.B., FERNANDEZ, E.B., LARRONDO-PETRIE, M.M. AND HASHIZUME, K. 2013. Whitelisting firewall pattern (WLF). In *Proceedings of the 20th Conference on Pattern Languages of Programs*, Monticello, Illinois2013 The Hillside Group, 2725683, 1-6.

YOSHIZAWA, M., WASHIZAKI, H., FUKAZAWA, Y., OKUBO, T., KAIYA, H. AND YOSHIOKA, N. 2016. Implementation Support of Security Design Patterns Using Test Templates. *Information 7*.