

Relating Security Requirements and Design Patterns

Reducing Security Requirements implementation impacts with Design Patterns

Felipe Silva Ferraz

Informatics Center
Federal University of Pernambuco
Recife, Brasil
fsf3@cin.ufpe.br

Rodrigo Elia Assad

Informatics Center
Federal University of Pernambuco
Recife, Brasil
rea@cin.ufpe.br

Silvio Romero Lemos Meira

Informatics Center
Federal University of Pernambuco
Recife, Brasil
srlm@cin.ufpe.br

Abstract— It is historically known that extensive phases of rework affect directly both costs and quality of already developed applications. In regard to requirements, systems security figure among those which are left behind, generating undesired steps of reformulation. This article is a short presentation on design patterns and non functional security requirements; it relates both and shows that an initial use of certain patterns, in early development phases, might reduce the impacts of relegating the systems security requirement implementation to a second plan.

Keywords-*design pattern; security; system engineer.*

I. INTRODUCTION

Many studies on Design Patterns show that the quality of systems which uses Object Orientation, is positively affected by them. However, it is important to remember that Design Patterns does not always make a good software architecture [4], actually Patterns will aim you to build those [3].

In quality field, security has brought attention as one of the most common subjects. Mainly systems are developed without security requirements in mind, which happen because developers usually tend to concentrate their efforts in first understanding systems functional requirements, putting non function ones, like security, on a second plan. In one of the many techniques used by developers, they focus in prototype construction to make how the system will work more comprehensible to the user, and use that prototype to better understand their needs[3]. The main issue about this technique is that non functional requirements are neglected, left for future implementation. Such problem becomes clearer on last phases of development, more precisely when it is time to deploy the system, which is when the team realizes that they still have to improve the project security. When that happens, team notices that implementing security is much more complicated and delicate process, going far beyond adding a new screen with Password and ID[3][13].

The main focus of this work will be the study of a group of design patterns which can not be found among the *Security Design Patterns* described by Schumacher in his book[2]. We intended to create a relation between those patterns and Security Requirements to reduce the impacts of a common problem, i.e., developers letting security requirements to be implemented on latter development phases [3]. This paper shows a synthetic way of getting a system prepared to have security requirements implemented on it. It is important to notice that the relation presented here

acts like a basic solution to a common problem. This paper is presented as follows: Section II will present a brief survey on design patterns and its categories. This next section will present a brief survey on information security and non functional security requirements, in this Section IV, a relation between security requirements and design patterns will be proposed; there will also be an explanation concerning the details of this relation. Finally, conclusion and future work are presented in Section V.

II. DESIGN PATTERNS

The enthusiasm of systems developers around design patterns is almost unquestionable since the publication of the famous book, *Design Patterns – Elements of Reusable Object-Oriented Software*, by the Gang-of-Four (GoF) [5]. Developers from around the world united themselves around the patterns idea, knowing for sure that those will provide them and their systems with elegance, directness, and versatility. They brought the benefit of resilience to changes, reducing any major damages to the system that might be cause by re-design. Through several characteristics related to maintainability[17], patterns had made their path into many projects[2].

Gamma et al.[5] state “*Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them*”, facing the great number of possibilities, an easy way to organize and categorize them was necessary; the first criterion used to classify patterns is purpose, this one reflects what the pattern does. According to it, design patterns can be Creational, Structural or Behavioral[5].

A. Creational Patterns

Creational Patterns are responsible for abstracting objects creation process. They are used to make the system independent of how the objects are created, represented or made. An entity that represents the Creational Pattern will use OO principles like inheritance to make different objects to be used in the system [5].

There are two main themes when dealing with Creational Patterns. The first one encapsulates the knowledge about which Class the system uses for real when creating an Object. On the second one, we have design patterns hiding how object instances are created and organized to be used by the rest of the system. In a certain way, to the system, the

only known entity is the defined interface, also known as a contract. Therefore, Creational Patterns provide a bigger flexibility regarding what is created, who creates, how it is returned and when.

This way the system returns and uses objects will be made of values and behaviors that may be easily changed during the development phase.

Example of Creational Patterns [5]:

- **Abstract Factory:** Uses an interface to create a family of objects without needing a concrete implementation.
- **Builder:** Keeps a complex object construction from its representation so that the construction could be reused for other representations.
- **Factory Method:** Defines a mechanism to create an object leaving for the factory the decision of which implementation to use.
- **Prototype:** Defines object types to be created using a default instance, makes new objects by copying the prototype.
- **Singleton:** Guarantees that only one instance of a specific entity is available in the system.

B. Structural Patterns

The objective of Structural Patterns is to define how classes and objects are combined in a way to create bigger and more complex systems. A simple sample is how to combine two different classes that extends from a third class. The result is a class that combines the properties from both original classes; conceptually, this pattern is useful to make different and independent class can work together [5].

Looking beyond design patterns, work with structures means to create new functionalities just using different behaviors of existent classes.

Some Structural Patterns [5]:

- **Adapter:** Converts an interface of a specific class into another. An *Adapter* transforms a class in another one which can be used by the client.
- **Bridge:** Keeps an abstraction of its implementation so that two different parts can work independently from each other.
- **Composite:** Organize objects in a tree structure to represent relation 1-to-many.
- **Decorator:** Adds responsibilities dynamically to an object.
- **Facade:** Provides a unique, and unified, interface to a set of systems interfaces.

C. Behavioral Patterns

The main focus of Behavioral Patterns are the algorithms, responsibility attribution and behavior between objects. They do not describe just objects and classes, but they also describe a way to make those objects communicate with each other. Those patterns represent complex flows of control that

are usually too hard to be followed during compilation time. They bring developers focus to the way that those objects connect themselves [5].

Example of Behavioral Patterns [5]:

- **Iterator:** Provides a way to access elements inside a set of objects without exposing their implementation.
- **Observer:** Defines a relation of 1-N (one-to-many) between objects, so that, when a specific action happens with one object, it has a way to notify N objects.
- **Command:** Encapsulates requests to an object, thus allowing the developer to parameterize the client with different kinds of requests.
- **Memento:** Without encapsulation breaking, captures and returns, to the exterior, the internal state of an object.
- **Strategy:** Defines a family of algorithms, encapsulates each one of them, and make them interchangeable. *Strategy* lets the algorithms change depending on the client.

Originally the GoF book presents patterns divided by two different criteria, Purpose and Scope. For the vision, we are going to present about security requirements, we choose to introduce only the Purpose criterion because it relates more synthetically with the concept we used.

III. LOOKING THROUGH SECURITY

Systems infrastructure format and characteristics have been evolving quickly. Therefore terms like “Internet Security” are unable to provide real world meaning for the mentioned structure [11]. Looking to a scenario with so many changes, information protection against non authorized changes is an important requirement that has gained attention, mostly by the increase on systems value, in which the information is so important that any non authorized access can generate company lost [6] through confidential information’s leak, and any data change that might lead to any fraudulent actions [7].

This new information paradigm brings more than just facilities and unrestricted access to data, it also brings new risks. Among them, the risk of valuable information being lost, stole, changed or used for bad purposes by someone. An artifact electronically stored, and available through a computer network is more vulnerable to an attack than the same information in printed paper and stored on a locker inside any office. Intruders don’t even need to break security physical mechanism of the office and break in, nor they need to be in the same country or even continent, they are able to access it without have to touch a single paper sheet [7].

Usually security aspects aren’t defined until the implementation finds itself on its last stages, therefore resulting in an expensive and hard attempt to correct some problems on a project still under construction [13].

Detect possible security requirements on initial phases of a project, and deal with them, may help to guarantee the product quality on final stages of its development.

A. Security Requirements

Requirements engineering inside a business, system, applications and components is much more than just documenting the present functional requirements. Even though the majority of the analysts use their time to understand requirements quality like: interoperability, performance, portability and usability, still many of them fail when the requirement is security related.

The reason for it is that requirement analysts has no knowledge about security issues, or security area, the few who has some kind of training had only a short and general vision about some security mechanisms like password and cryptography instead of some lessons, or sections, about real requirement in this area [10]. It is common to find references for security requirements as a security mechanism description, for example ISO 15408, PKI, and others. This kind of approach say what the requirement is to do but is not clear about why it has to do it [14].

Security requirements deals with how goods of a system must be protected against any kind of evil or threat [14][13]. A good of a system is something, in system context, touchable, or not, that must be protected [15]. An evil, or threat, that the system must be protected from, is a possible vulnerability that might affect a good. Vulnerability is a system weakness which an attack tries to explore. Security Requirements are functional requirement restrictions with the intent to reduce vulnerabilities scope [13], must be expressed in terms of business goal, or the relevance of security itself for the business[16].

Donald Firesmith resumes the most commonly Security Requirements in a really simple way, like [10]:

- **Identification Requirements:** Defines any requirement that specifies to the extent which an entity knows, recognize, the external parts of the system (users, external applications or services), before interact with those. E.g.: Application must be able to identify all of its clients.
- **Authentication Requirements:** Defines any requirement that specifies to the extent which an entity verifies, confirms or denies the entity presented by external parts (users, external applications or services), before interact with those. E.g.: Application shall verify the identity of its users.
- **Authorization Requirements:** Defines any requirement related with access and privilege that a specific entity will gain after the authentication and validation phases. E.g.: Application must be ready to allow each client to access his or her account information.
- **Immunity Requirements:** Defines any requirement that specifies the extent which an entity protects itself from invasions, infections or changes, by malicious programs, such as: Virus, Trojans, worms and scripts. E.g.: Application shall protect itself from

infection or related problems checking all downloaded or inserted data.

- **Integrity Requirements:** Defines any requirement that specifies the extent in which a communication, or data, hardware or software, protect themselves of corruption attempts by others. E.g.: The application shall be able to prevent any unauthorized entity to corrupt its data.
- **Intrusion Detection Requirements:** Defines any requirement that specifies the extent in which an, non authorized, attempt to access or modification is detected, registered and notified, by the system. E.g.: application shall detect all unauthorized access and report it to the responsible for it.
- **Nonrepudiation Requirements:** Defines any requirement that specifies the extent where a part of a specific system can prevent a third party of the system to deny its relation to a specific action. E.g.: an exchange of message. E.g.: The application shall be able to store proofs of all accesses or actions.
- **Privacy Requirements:** Also known as Confidentiality. Defines any requirement that specifies the extent in which important data and communications are kept private from accesses coming from other individuals and programs. E.g.: The application can not store or post any personal data of its users.
- **Security Auditing Requirements:** Defines any requirement that specifies the extent that a team, responsible for it, can verify the status and use of security mechanism through some kind of analysis related to a specific or a group of events. E.g.: The application shall store a set of particular information about actions made, and make a future search possible.
- **Survivability Requirements:** Defines any requirement that specifies the extent in which an entity keeps doing its mission, providing essential resources to the users even in the presence of an attack. E.g.: The application can not have failure points.
- **Physical Protection Requirements:** Defines any requirement that specifies the extent in which an entity protects itself physically from other entity. E.g.: The application location shall be protected from physical damage.
- **System Maintenance Security Requirements:** Defines any requirement that specifies the extent in which the system must keep its security definitions even after modifications or upgrades. E.g.: The application security must be kept after any upgrades.

IV. CONNECTING DESIGN PATTERNS AND SECURITY REQUIREMENTS

Initially we presented the classification proposed by the GoF for the known design patterns, focusing on the purpose division, still looking to that classification; we exemplified

some patterns of each category and their usage. After the Section related to Security Requirements we were able to verify that some of those requirements, fit on a conceptual level, in one of the classifications used for the patterns.

From the characteristics showed in Section III for each security requirement, we are now going to divided them in three groups, called as Creation, Structure and Behavior, grouping each one of the according to their purpose on security area.

The following Table I shows a survey of the mentioned division:

TABLE I. REQUIREMENT DIVISION ACCORDING TO SPECIFIC CHARACTERISTICS.

	Purpose	
Creation	Structure	Behavior
Identification	Immunity	Intrusion Detection
Authentication	Integrity	Security Auditing
Authorization		
Nonrepudiation		
Privacy		
	System Maintenance Security	

Since Physical Protection and Survivability Requirement deals with problems related physical questions focusing on an environment that must be protected, those requirement are not part of the mentioned structure.

It is important to remember that the division proposed above represents a mechanism to help developers, on initial phases of a project, to solve the problem detected by Yoder, that problem references to developers letting security requirements implementation only to the end phases of a project [3].

More than just a division, the next sections will present a solution where, through the creation of stubs responsible for security requirements, we will find an easy and practical way to implement security's rules, specifications, policies and definitions even in the last development steps, this will be responsible in some cases for increasing the projects prices and costs [3], the expensive rework phase will be minimum, since the security structure will be ready.

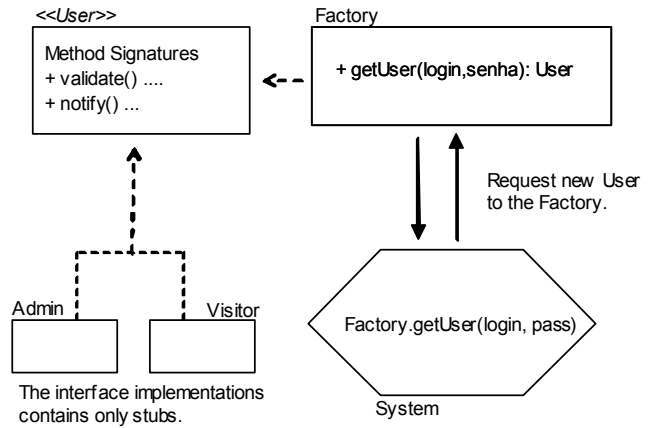
For each one of the requirements, from the categories, we have:

A. Creation

The requirements related to any kind of entity creation are classified in this category. Identification, Authentication and Authorization Requirements deals with the creation of entities responsible for: identifying an actor or system; the validation of this actor or system; and for set or remove permission that those entity has.

In a first implementation, the entities created by this pattern will be returned as an empty entity, an implementation that contains only a stub, for example: Factory, method or abstract, see Figure 1.

Figure 1. Factory Schem for an user creation.



The Factory represented above has also an interface definition and must be able to return as many implementation of that interface as possible, however those implementations, since they are just an initial entity, must be created empty, just as a stub, returning *true* to every method that represents a checking Figure 2.

Figure 2. Implementation Sample.

```

public class Admin implements User{

    public boolean validate() {
        // TODO: Modify this block,
        //adding the correct user validation
        return true;
    }

    public boolean notify() {
        // TODO: Modify this block, adding
        // the correct notification to a user
        return true;
    }

}

```

Nonrepudiation and Privacy Requirements will use the same objects created in the Factory to guarantee both that the user can identify precisely the responsible for each action and keep its privacy, in a way that only a correct user, with right permissions, can have access to a specific area.

B. Structure

The requirements related to systems structure are classified in this category. They are: Immunity Requirement and Integrity Requirement.

More than just explicit how systems structures will be arranged, or how classes can be adapted, this category classifies requirements where entities will use information already created by other objects.

Those objects contain information regarding the user identity and permissions and must be used to guarantee the immunity and integrity requirements.

First of all, Immunity Requirement must provide a way where only the user, or entities, with permission for it, can have access to the system, or environment, it could be

reached through checking the object that represents the user and its permission, the same object must be always verified every time an access to some data or hardware is made, this particular action will work as a functionality responsible for the Integrity Requirement, see Figure 3.

Figure 3. Checking users permission.

```
if(user.hasPermission(User.WRITE)) {
    //Execute the specific code.
}
```

Note that in this case, since the implementation of the interface has been created with the correct coding, is always returning true; the main functionality of the system can be verified, letting security policies to be implemented only on the last phases of the development.

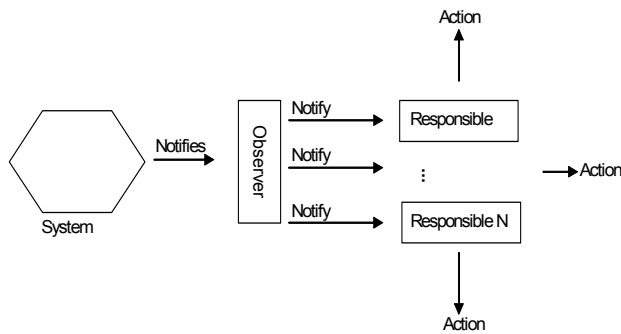
C. Behavior

Requirements related to systems behavior are classified under this category. Intrusion Detection and Security Auditing Requirements focus on receiving systems notifications to a set of events, in the first, events related to faults or intrusions, in the second one events related to a bigger set of possibilities that initially detected and then stored for a future search. The First requirement notifies an entity, or entities, when an intrusion is detected, Auditing requirement organizes for future query any system action or interaction.

For this kind of situation, the proposal is to build a stub that implements a pattern related to a behavioral pattern. E.g.: Observer.

The Observer must be notified in every necessary case like: every time a specific search or insertion happens like in Figure 4.

Figure 4. Observer notify schema.



From those notifications the Observer will notify all entities, registered on it. All entities interested in receiving a notification in a case of a specific action must only implement the correct interface and register itself to the correct Observer.

Once registered to the Observer, the registered entities will receive notifications so they can execute some system's

action, like: veto access, restrain use, restart system, or even just send a message to administration team.

Initially, the actions mentioned in this section could be implemented empty methods, stubs of the desired interface.

D. System Maintenance Security Requirement.

Finally let us give a look on a bigger requirement. According to Firesmith's definition [11], we realize that this requirement is responsible for the deployment of a new system version, an upgrade; it says that the systems definition must be kept and unchanged. Through use of some patterns related to Creation, Behavior and Structure categories we can have an easier code update, knowing exactly where each requirement affects, and by that knowing precisely where to change without compromising the rest of the system, letting correct entities untouched, changing only what is really necessary.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a basic solution for implementing security requirements. As we discussed, there is a big negligence related to security issues, most of them happen because of misunderstanding of security terms or just missing some details.

Through the technique presented, we were able to build an adequate system to security requirements, only by using a well known approach, Design Patterns[5]; using those we expect to bring security requirements closer to developer and easily implemented, even when they are still not completely defined.

And since the relation presented here is a result of some researches executed on an already running project, a future work is to apply the technique mentioned in some small and quick projects, so that a quantification and estimate related to the quality and time can be extracted for future reference.

Table II shows, in a syntactic representation, the relationship created by security requirements, suggested pattern and their use.

TABLE II. PATTERNS X REQUIREMENTS X USE

		Requirements	Sample of Pattern to be used	Use / Impact
Creation	System Maintenance Security	Identification	Factory Method or Abstract Factory.	Instance a Stub to identify, authenticate and add all permissions to all users.
		Authentication		
		Authorization		
		Nonrepudiation		
Privacy				
Behavior	Immunity	_____	Make the entire system to use the objects created by the creation pattern. Check those objects before the correct actions to be made.	
	Integrity			
Structure	Intrusion Detection	Observer	Always notify the Observer that specific events had happened. And let it to take correct providence. Initially those actions will be empty.	
	Security Auditing			

REFERENCES

- [1] Dhillon, Gurpreet, Principles of Information Systems Security Text and Cases, Willey, 2006.
- [2] Schumacher, Markus, Security Patterns Integrating Security and System Engineering, Willey, 2006.
- [3] Yoder, Joseph and Barcalow, Jeffrey, Architectural Patterns for Enabling Application Security, 1997.
- [4] Khomh, Foutse and Guéhéneuc, Yann-Gaël, Do Design Pattern Impact Software Quality Positively, 2008.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison - Wesley, 1st edition, 1994.
- [6] Menezes, A., vanOorschot P. and Vanstone S., Handbook of Applied Cryptography, by CRC Press, 1996.
- [7] Ferraiolo, D.F., Cugini, J., Kuhn, D.R. Role Based Access Control: Features and Motivations. In: Computer Security Applications Conference, 1995.
- [8] Thomas A. Longstaff. James T. Ellis, Shawn V. Hernan, Howard F. Lipson, Robert D. McMillan, Linda Hutz Pesante, Derek Simmel Security of the Internet, 1997.
- [9] McDaniel, G. Ed. IBM Dictionary of Computing. New York, NY: McGraw-Hill, Inc., 1994.
- [10] Firesmith, Donald, Engineering Security Requirements, in Journal of Object Technology, vol. 2, no. 1, January-February 2003, pages 53-68. http://www.jot.fm/issues/issue_2003_01/column6
- [11] Firesmith, Donald, Analyzing and Specifying Reusable Security Requirements, 2004.
- [12] Li Gong, Java Security: Present and Near Future IEEE Micro, 17(3):14--19, May/June 1997.
- [13] Haley, Charles, Laney, C. Robin, Nuseibeh, Bashar, Deriving Security Requirements from Crosscutting Threat Descriptions
- [14] Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar Nuseibeh. A framework for security requirements engineering. In SESS'06, Shanghai, China, May 2006. ACM.
- [15] ISO/IEC. Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 1: Introduction and General Model, 15408-1. Geneva Switzerland: ISO/IEC, 1 Dec 1999.
- [16] Ivan Tirado, Business oriented information security requirements development. In ACM, New York, NY, USA, 2008.
- [17] Lerina Aversano , Gerardo Canfora , Luigi Cerulo , Concettina Del Grosso , Massimiliano Di Penta, An empirical study on the evolution of design patterns, Proceedings of the the 6th ACM SIGSOFT symposium on The foundations of software engineering, 2007.