Security Patterns

Ronald Wassermann and Betty H.C. Cheng*
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824, USA
Email: \{wasser17,chengb\}@cse.msu.edu

*Please contact B. Cheng for all correspondences.


## Contents

1 Introduction ........................................... 1

2 Background ............................................. 3

2.1 Viega’s and Mcgraw’s ten principles ................. 4

2.1.1 Principle 1: Secure the weakest link. .......... 4

2.1.2 Principle 2: Practice defense in depth. ....... 4

2.1.3 Principle 3: Fail securely. ...................... 5

2.1.4 Principle 4: Follow the principle of least privilege. .. 5

2.1.5 Principle 5: Compartmentalize. ................ 5

2.1.6 Principle 6: Keep it simple. .................... 6

2.1.7 Principle 7: Promote privacy. .................. 6

2.1.8 Principle 8: Remember that hiding secrets is hard. .. 7

2.1.9 Principle 9: Be reluctant to trust. .............. 7

2.1.10 Principle 10: Use your community resources. ... 7

2.2 UML Notations ....................................... 8

2.3 The Pattern Approach ................................ 9

3 Security Patterns ..................................... 11

3.1 Security Pattern Definitions ....................... 11

3.2 Security Pattern Template ......................... 11

4 Examples of Security Patterns ....................... 16

4.1 Single Access Point ................................ 18

4.2 Check Point ......................................... 27

4.3 Roles ............................................... 36

4.4 Session ............................................. 43

4.5 Full View With Errors ............................. 49

4.6 Limited View ...................................... 53

4.7 Authorization ...................................... 57

4.8 Multilevel Security ................................. 62
List of Figures

1  Traceability in diagrams ................................. 17
2  UML class-diagram: SAP Example: Online store ..................... 20
3  UML class-diagram: SAP Example: Medication system .............. 21
5  UML sequence-diagram: Single Access Point: Recipient not available .......... 23
6  UML sequence-diagram: Single Access Point: Example 01 .............. 23
7  UML sequence-diagram: Single Access Point: Example 02 .............. 24
8  UML class-diagram: Check Point ................................ 28
9  UML sequence-diagram: Check Point: access granted ................ 30
10 UML sequence-diagram: Check Point: access denied ................ 30
11 UML sequence-diagram: Check Point: Example 01 .................. 31
12 UML sequence-diagram: Check Point: Example 02 .................. 31
13 UML state-diagram: Check Point ................................ 32
14 UML class-diagram: Roles .................................. 39
15 UML class-diagram: Session .................................. 45
16 UML sequence-diagram: Session: Example ......................... 46
17 UML class-diagram: Authorization (In accordance to Fernandez [5]) .......... 58
18 UML sequence-diagram: Authorization approved .................... 59
19 UML sequence-diagram: Authorization rejected .................... 60
20 UML class-diagram: Multilevel Security (In accordance to Fernandez [5]) .......... 63

List of Tables

1  Canonical Pattern Template versus GoF Template .................... 10
2  Security Pattern Overview .................................. 16
1 Introduction

During the process of software development expert knowledge in the domain of the implemented system is required. Furthermore, in today’s systems with various communication features, security considerations are of greater interest than ever. Therefore, a fair amount of additional security expertise is needed to meet non-functional security requirements. A common approach to overcoming knowledge gaps among developers is to use patterns (analysis patterns [6], design patterns [7], specification patterns [4], etc.). In recent years, pattern-based approaches to software development, applied to different domains, have received significant attention in the software engineering community. In the security domain, it is challenging to capture and convey information in order to facilitate security, which is a very abstract goal. In this report we give a collection of security patterns that have been identified by the community. We use a variation of the design pattern template [7] that better suits the presentation of security-specific information in order to facilitate reuse of security knowledge.

Providing expertise that significantly improves system development with respect to security is an ambitious goal. In contrast to functional requirements that have a concrete solution, security is difficult to measure and highly dependent on the environment. Other pattern-based approaches like the well-known Design Patterns from Gamma et al. [7] are believed to greatly enhance productivity\(^1\) of the software development process. They convey expertise. Unfortunately, the structure provided by various pattern templates is not sufficient to portray all security relevant aspects. Some approaches [5][11][16] that apply patterns to the field of security use the regular or a slightly modified Design Pattern template. The enhanced Security Pattern Template\(^2\) presented herein contains additional information, including behavior, constraints and related security principles, that addresses difficulties inherent to the design of security critical systems.

The security features that a system needs are highly dependent on the environment in which the system is deployed. As the pattern approach is not capable of fully covering all possible constellations of security, it is crucial that a developer is provided with information that enables an

---

\(^1\)the Design Patterns book of Gamma et al. [7] received an award for software development productivity in 1994.  
\(^2\)see Section 3.2
evaluation of the situation that will lead to the selection of appropriate patterns. By introducing and connecting general security principles with a pattern’s substance, the developer gains security insight by reading and applying the pattern. Furthermore, behavioral information and security-related constraints are added in this new pattern template. The developer can use this information to check if a specific implementation of the pattern is consistent with the essential security properties.

Our augmented Security Pattern Template enhances the communication of security-specific knowledge that is related to a concrete application. Furthermore, it promotes the verification of security-relevant properties. Hence, overall security can be improved during the design stages of development by applying this pattern-based approach.

The remainder of this document is organized as follows. Section 2 gives background information for this work including general security principles, a brief description of some UML diagrams, and an overview of pattern-based approaches. The Section 3 defines Security Patterns and introduces our template. Finally, Section 4 illustrates in several examples how the pattern template is applied.
2 Background

This section describes previous work and fundamentals that are relevant to this report. We start with an introduction of ten principles [14] that provide guidelines to more secure system development. The descriptions of *Security Patterns* reference those principles. Next, the selected UML notations that are used in the *Security Patterns* section are briefly overviewed. Finally, we provide a historical perspective of pattern-based approaches that elucidate the pattern approach, especially *Design Patterns*, and explain its application to this work.
2.1 Viega’s and McGraw’s ten principles

To improve development of secure software Viega and McGraw [14] point out ten guiding principles to achieve better security. They state, in contrast to checklist based approaches, the use of guiding principles can help to cope with unknown attacks. Although guidelines do not guarantee security, their application can help prevent common errors during the software development process. Some of the principles exhibited by Viega and McGraw are based on Saltzer’s and Schroeder’s [10] eight design principles, which were published in 1975.

The ten principles, outlined in the following section, can facilitate the understanding of particular Security Patterns and give security insight. Therefore they are used in our pattern-based approach and will be referred to as the Ten Principles.

2.1.1 Principle 1: Secure the weakest link.

An intruder will usually attack those parts of a system that are most likely to break. For that reason, the level of security in a software system is determined by its weakest component. In order to improve system security possible weaknesses must be identified and strengthened until the risk of violations can be considered acceptable. Viega and McGraw [14] indicate that a system’s users and administrators can be a major vulnerability too. They might easily become victims of social engineering\(^3\) if the current security policy does not take account of those attacks.

2.1.2 Principle 2: Practice defense in depth.

An exception to the first principle are overlapping security mechanisms. If more than one protective measure exists in a system, then the level of security is not necessarily determined by the weakest part. Every additional protection layer may contribute to system’s security. Viega and McGraw [14] propose in their second principle the use of several security layers to create a more effective defense against attacks.

\(^3\)Social engineering is a non-technical kind of attack that makes use of human weaknesses. An exemplary scenario might consist of an intruder contacting authorized users in order to make them reveal information that compromises the security [15].
2.1.3 Principle 3: Fail securely.

Security flaws often come along with system failures. Unfortunately failures cannot be avoided completely in complex software systems. Thus, Viega and McGraw [14] point out that it is even more important to plan failure modes and assure that a system’s security is not compromised by exceptional behavior. Some systems perform insecure operations during failure modes in order to provide certain functionality or to maintain compatibility with old standards. This common practice of code violates the 'fail securely'-principle. An attacker might find a way to trigger the insecure system failure and take advantage of its behavior.

2.1.4 Principle 4: Follow the principle of least privilege.

In 1975, Saltzer and Schroeder [10] published the principle of least privilege in their paper on protection of information in computer systems. It states that every entity in a system should be granted only the minimum set of permissions that are needed to perform its designated tasks. As it is not practicable to completely prevent misuse of rights, application of this guideline helps to limit the impact on the systems affected. When Viega and McGraw [14] adapted this principle, they remarked that many programmers violate it just for the reason of laziness. It is much more convenient, and also much more insecure, to grant a program full access than to figure out in which manner system’s resources need to be accessed and to restrict their use.

2.1.5 Principle 5: Compartmentalize.

The principle of compartmentalization aims for a similar goal as the principle of least privilege: it tries to minimize the damage that an attack can cause. Viega and McGraw [14] recommend segmenting a system into several components that can be protected independently. Thus, a security breach in a smaller entity would not affect other parts of the system. Unfortunately, those systems that protect their parts from each other are more difficult to program and require more administrative work than others.

According to principle four: *Follow the principle of least privilege*, compartmentalization can provide a finer grain for granting access rights. This allows a more restrictive access model to be implemented.
2.1.6 Principle 6: Keep it simple.

Viega and McGraw [14] promote the idea that one should keep a system as simple as possible in order to avoid unnecessary complexity. This objective is fairly simple and not new to the field of computer science. It is obvious that the required effort to understand a system grows with a system’s complexity. Saltzer and Schroeder [10] named this principle *economy of mechanism*. Furthermore, they point out that undesirable access paths can be discovered more easily in a plain, simple system.

Viega and McGraw state that usability is a significant part of simple design. With respect to security, they consider it even more important than in any other domain, because many security-relevant actions depend on users’ decisions. Saltzer and Schroeder take into account this aspect in their principle of *psychological acceptability*. They state that the human interface has to support the application of protection mechanisms.

Some principles exhibited here contradict each other if they are applied to the greatest possible extend. Thus, it is necessary to evaluate to which extend the application of a principle is reasonable according to your specific system. If the ultimate goal is simplification then defense cannot be practiced in depth, because it would increase the system’s complexity. Finding an appropriate combination of the principles can be a challenging task if a system of good security is to be designed.

2.1.7 Principle 7: Promote privacy.

Promoting privacy consists of two main objectives. First, the amount of information that can be gathered about a system and its users should be minimized. This objective will make the system a harder target and it protects the users privacy. Second, make use of misinformation to mislead attackers. If they rely on wrong assumptions then its more unlikely that they can do harm.

Viega and McGraw [14] point out that there is a tradeoff between holding back information and usability. A user might have to remember data for security purposes. It is inconvenient for him to reenter it.
2.1.8 Principle 8: Remember that hiding secrets is hard.

Usually a system’s security depends on certain secrets being kept. Once private keys, passphrases or secret algorithms are revealed, security can be easily compromised. It can be a difficult task to protect such secrets. Even critical information in binary form does not protect it from being uncovered. Some attackers have sufficient reverse engineering capabilities to analyze and understand machine-readable code. Viega and McGraw [14] suggest being suspicious even of known entities and to take into account insider attacks.

2.1.9 Principle 9: Be reluctant to trust.

Some security violations are possible, because system developers extend trust unnecessarily. A more pessimistic point of view during system design can help to recognize weak points. Viega and McGraw [14] advise programmers to design systems whose parts mistrust each other. Furthermore one should not rely on the security of off-the-shelf solutions but rather practice defense in depth. Another design flaw is hiding secrets in client code. Skilled users might be able to extract information they should not have and abuse that knowledge.

2.1.10 Principle 10: Use your community resources.

According to Saltzer’s and Schroeder’s [10] principle of open design, Viega and McGraw [14] state that community resources usually are more secure than routines written by individuals. Widely distributed programs have been executed and tested many times and therefore it is very unlikely that unknown errors and security weaknesses exist. Saltzer and Schroeder point out that decoupling the protection algorithms from keys enabled public scrutiny of protection mechanisms, thereby potentially improving their overall capability. Furthermore the protection of the key is more practicable than keeping an algorithm secret.
2.2 UML Notations

The final version of this report will include brief descriptions of different notations including UML class, sequence and state diagrams.
2.3 The Pattern Approach

This section clarifies what patterns are and why they are useful. After a brief presentation of the pattern approach we focus on design patterns and the different types of templates that are mainly used. Finally we take a look at Anti-Patterns.

The first person, who used the pattern approach was Christopher Alexander [1]. In his book: *A pattern language: towns, buildings, construction*, he gave a succinct definition of what a pattern is:

> “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

*Christopher Alexander [1], Page x*

Even if Alexander meant to describe the architectural problems that accompany the planning and construction process of towns and buildings, his brief definition is applicable to many different areas. The computer science community finally became aware of patterns as means of coping with recurring problems, when Gamma et al. (who are also referred to as the *Gang of Four (GoF)*) [7] published their book *Design Patterns: Elements of Reusable Object-Oriented Software*, in 1994. Since then, the pattern approach has been applied in many different realms. Fowler [6] stated, while presenting his book on *Analysis Patterns*, that the GoF’s book had more influence on software patterns than Alexander’s ideas.

In recent years, the pattern approach obtained increasing popularity. Several factors account for the acceptance among members of computer science community. Firstly, patterns improve and simplify communication among people who know the pattern by extending the existing terminology. A pattern name will be associated with a specific problem and its solution. Secondly, expert knowledge of a specific domain can be captured in a pattern. Structures that experienced designers intuitively apply can be communicated to novices. Thirdly, the use of patterns unifies design and thus improves comprehensibility.

Gamma et al. transferred Alexander’s patterns to describe solutions to software design problems in terms of objects and coined them *Design Patterns*. The GoF point out four mandatory elements of a *Design Pattern*: pattern name, problem, solution and consequences. Although those sections embody the core of a pattern the captured information can be structured in various ways. Thus,
several Design Pattern formats can be found. The most common templates are the one presented by Gamma et al. [7] and the canonical template, which is used by the AG Communication Systems (AGCS) [13]. Table 1 gives an overview of the different sections of those formats and how they relate to each other. It is based on information that was presented by Brad Appleton [2] in his report on Patterns and Software: Essential Concepts and Terminology. The different elements that we adapt into our Security Pattern Template will be explained when used in Section 3.2.

Unlike regular patterns that point out desirable solutions to recurring problems, Anti-Patterns depict common design errors and pitfalls. They describe structures, that are unfavorable for certain reasons. Besides just indicating unsuitable structures Anti-Patterns can also give advice how to avoid or correct shortcomings in design. Similar to regular patterns, Anti-Patterns capture experience.

<table>
<thead>
<tr>
<th>Canonical Template</th>
<th>GoF Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name, Classification</td>
</tr>
<tr>
<td>Aliases</td>
<td>Also Known As</td>
</tr>
<tr>
<td>Problem</td>
<td>Intent</td>
</tr>
<tr>
<td>Context</td>
<td>Applicability</td>
</tr>
<tr>
<td>Forces</td>
<td>Motivation</td>
</tr>
<tr>
<td>Solution</td>
<td>Structure, Participants, Collaborations, Implementation, Sample Code</td>
</tr>
<tr>
<td>Resulting Context</td>
<td>Consequences</td>
</tr>
<tr>
<td>Rationale</td>
<td>-</td>
</tr>
<tr>
<td>Known Uses</td>
<td>Known Uses</td>
</tr>
<tr>
<td>Related Patterns</td>
<td>Related Patterns</td>
</tr>
<tr>
<td>Sketch</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Canonical Pattern Template versus GoF Template
3 Security Patterns

This section is organized in two subsections: Section 3.1 defines terminology that is inherent to Security Patterns. Section 3.2 presents our Security Pattern Template.

3.1 Security Pattern Definitions

In the final version of this report the terms Security Pattern and Security Pattern System will be defined here. Furthermore, we will elucidate previous work in the area of Security Patterns.


Security Pattern:

A Security Pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution.

Security Pattern System:

A Security Pattern System is a collection of security patterns, together with guidelines for their implementation, combination and practical use in security engineering.

3.2 Security Pattern Template

The following template for Security Patterns is derived from the template for the well-known Design Patterns from Gamma et al. [7]. Some sections have been altered to convey more security-relevant information than the original template. Other parts are completely new (Behavior, Constraints, Supported Principles, Community Resources) and provide additional information that has not been captured or been relevant to common design patterns. The approach depicted below tries to facilitate an efficient way to reuse security specific knowledge.

- Pattern Name and Classification

In accordance to Gamma’s template, the Name is a primary key\(^4\) to the pattern. It should

\(^4\)Usually primary key is a term used in database design. In a relational table the primary key unambiguously identifies each record (row) of a table. We use this term in a similar fashion for identifying patterns.
be self-explanatory and intuitive in order to improve communication among designers.

- **Intent**
  The intent section describes briefly for what this pattern is used. It names security problems that can be solved by application of this pattern. It may also introduce the pattern as an elegant way of improving a system according to the *Ten Principles*.

- **Also Known As**
  If synonyms exist for this pattern then they should be named in this section.

- **Motivation**
  This part describes security problems (or violations of the *Ten Principles*) that are addressed by application of this pattern. Furthermore, the motivation section should show the basic ideas of the pattern in an illustrative way and convey how the mentioned problems are solved. This could be done by giving an appropriate example of the patterns application.

- **Applicability**
  Describes the context in which the pattern can be used. Under which circumstances should the pattern be applied? Does it address application-level, host-level or network-level security?

- **Structure**
  The structure section uses UML class diagrams to give an overview of the static components used in this pattern.

- **Participants**
  Describes the different classes or objects depicted in the structure section.

- **Collaborations**
  A textual description of the interaction among the participants and how they perform their different tasks.

- **Behavior**
  To illustrate the pattern’s behavior and interaction more formally, UML state and sequence diagrams are used to depict the dynamic aspects of this pattern.
• Constraints
The constraints section describes properties that must be fulfilled at all times by the implementation of the pattern. We enrich the template with constraints in order to facilitate verification tasks. The goal of this approach is to provide an efficient way to rigorously check system design and code during the process of software development. Our idea is that the application of the pattern leads to a formal model that can be used to automatically check your system against the properties presented herein.
In order to categorize the constraints, their function may be named along with them. Frequently occurring security-related classes of constraints are:

– Availability,
– Authenticity,
– Confidentiality,
– Integrity.

• Consequences
How does the pattern cope with the outlined problems. What further information is relevant for the application of the given pattern. How does it affect the current system and its security? Which side-effects arise from its use? Kienzle et al. [8] propose the following template as a means to categorize consequences. The Security Patterns outlined herein make use of this criteria to depict tradeoffs and outcomes systematically.
Accountability: According to the National Security Telecommunications and Information Systems Security Committee, accountability describes a system’s ability to determine the responsible source for activities [9]. How does the application of the pattern impact the system’s capability of accomplishing those tasks? How is, for example, logging or authentication affected?

Confidentiality: The assurance that information is not accessed by unauthorized parties is termed Confidentiality [9].

Integrity: Which impact has the application of the pattern on the protection of information against malicious modification or destruction.

Availability: Availability assures that authorized users can use a system’s resources when required [9]. How does the pattern support or limit availability?

Performance: Which impact has the pattern on a system’s performance? Does it slow it down or improve working speed?

Cost: Which expenses accompany the application of the pattern?

Manageability: How is the management and maintenance of the system affected?

Usability: Which changes coming along with the implementation of the pattern might be perceived by a user?

- **Implementation**
  What are issues of concern, when the pattern is implemented?

- **Known Uses**
  Which systems have implemented this pattern?

- **Related Security Patterns**
  Are there any other Security Patterns that may be used with this one? Is it part of a Security Pattern System?\(^5\)

- **Related Design Patterns**

\(^5\)The term Security Pattern System is defined in Section 3.1

14
Which *Design Patterns* can be used to realize the pattern? How can they contribute to system security? Which of the *Ten Principles* do they support?

- **Supported Principles**
  Which of the *Ten Principles* are supported by this pattern and which are violated (if any).

- **Community Resources**
  Names of libraries, exemplary implementations or other resources that support the application and implementation of the pattern, if any.
4 Examples of Security Patterns

This section contains examples of Security Patterns. Table 2 outlines the patterns that are described in this section. Similar to Design Patterns [7] we can classify them into the categories Creational, Structural and Behavioral patterns. Furthermore, their abstraction level defined in terms of Application-level, Host-level, and Network-level are shown. In order to give a high-level perspective on the purpose of each pattern, their relation to the Ten Principles is pointed out.

The patterns Single Access Point, Check Point, Roles, Session, Full View With Errors, Limited

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Classification</th>
<th>Abstraction level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Access Point</td>
<td>S AHN</td>
<td>x</td>
</tr>
<tr>
<td>Check Point</td>
<td>S AHN</td>
<td>x</td>
</tr>
<tr>
<td>Roles/RBAC</td>
<td>S AHN</td>
<td>x</td>
</tr>
<tr>
<td>Session</td>
<td>C A</td>
<td>x</td>
</tr>
<tr>
<td>Limited View</td>
<td>B A</td>
<td>x</td>
</tr>
<tr>
<td>Full View with Errors</td>
<td>B A</td>
<td>x</td>
</tr>
<tr>
<td>Security Layers</td>
<td>S A</td>
<td>x</td>
</tr>
<tr>
<td>Authorization</td>
<td>S AHN</td>
<td>x</td>
</tr>
<tr>
<td>Multilevel Security Pattern</td>
<td>S AHN</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2: Security Pattern Overview

View are based in part by patterns from Yoder and Barcalow [16] on Architectural Patterns for Enabling Application Security. Yoder and Barcalow focused on creating a framework that improves application development. Therefore, they presented the named patterns in canonical form⁶.

Our approach shows that some of their patterns may be applicable beyond the scope of application security. To convey more security-related information we use our augmented Security Pattern template introduced in Section 3.2 on page 11.

Fernandez [5] also introduced a few security-related patterns. His Authorization, Role-Based Access Control (RBAC) and Multilevel Security Pattern refer in part to Yoder’s and Barcalow’s work, mentioned above. As Security Patterns on a relative high abstraction level, Fernandez patterns are

⁶pattern format, explained in Section 2.3
a good addition to our overview.

Note that Fernandez’ RBAC and Yoder’s and Barcalow’s Roles describe the same pattern.

In order to show patterns of high abstraction levels, different types of diagrams may be required.

To determine whether a critical communication between two objects is accessible to the public, information on their deployment is needed. Especially if a pattern is at host or network-level we map objects and their nodes in a diagram to show their distribution. Figure 1 shows how the diagrams relate to each other.

Figure 1: Traceability in diagrams
4.1 Single Access Point

The Single Access Point (SAP) pattern was first used by Yoder and Barcalow in 1997 [16]. While they concentrated their work on a framework for application development, the basic idea for SAP is also appropriate beyond that scope. The concept of limiting extraneous access to a single channel in order to facilitate control, may be used in any self-contained system that communicates with others.

- **Pattern Name and Classification**
  The following Security Pattern is commonly known as Single Access Point. It provides a scheme for the static design of a system. Hence, it can be considered a structural Security Pattern.

- **Intent**
  The Single Access Point pattern defines one single interface for all communication with system external entities in order to improve control and monitoring.

- **Also Known As**
  The Single Access Point pattern is also referred to as [16]
  
  - Guard Door,
  - Login Window,
  - One Way In or
  - Validation Screen.

- **Motivation**
  Due to various access points, many systems cannot be protected effectively against attacks from the outside. Hidden back doors and different (inconsistent) implementations of security policies aggravate protection. The application of the Single Access Point pattern prevents external entities to communicate directly with components in the system. All inbound traffic is routed through one channel, where monitoring can be performed easily. Additionally, the Single Access Point is an appropriate place for capturing an information log on the parties currently accessing system. This data may be useful inside the system to verify certain access requests and to determine their rights.
• **Applicability**

The Single Access Point may be applicable to self-contained systems that need to communicate with external entities. It can be used at the application-level as well as at the host or network-levels. Even though implementation at the abstraction-level of application development is more apparent at first glance. Application at the network-level implies that all sub-nets inside the system’s boundaries are isolated from other nets. The only connection to the outside is a Single Access Point. While we assume virtual isolation of system internal entities in high abstraction levels, lower design levels have to include this goal in their models (for example by adding encryption, signing of messages, and tokens that guarantee freshness).

The system’s deployment structure determines where further securing effort is necessary.

Yoder and Barcalow [16] describe the following forces that result from the implementation of a Single Access Point. It is obvious that this pattern cannot be used with systems that need several entry points in order to provide greater flexibility. Furthermore, multiple access points alleviate adaptation in different environments. Each of them could be customized to request only the authorization information required to execute the current operation - a Single Access Point that grants access to several parts of the system might ask for information that will not be used during the session.

• **Structure**

The Single Access Point represents the system’s only connection to the outside. All incoming communication requests are passed to the Single Access Point instance. From there, they will be directed to the intended recipient, if all security-relevant properties are met. The UML class-diagrams depicting those structural relationships are shown in Figures 2 and 3. The rectangular box depicts the system’s boundaries. The class *Single Access Point (SAP)* is the only one that interacts with external entities. The first example (Figure 2) is a simplified model of an online store. Requests from outside (e.g. from known customers, a store manager or anonymous guests browsing the online store) are directed to the SAP and then forwarded to the internal class. An example message sequence could be a query that returns a set of available products in the store. The second example (Figure 3) depicts a medication information system that might be used in a hospital. Both application domains show how external entities have to request information from the system via the SAP.
Figure 2: UML class-diagram: SAP Example: Online store
Figure 3: UML class-diagram: SAP Example: Medication system
• **Participants**

  – **External Entities**
    * are components located outside the system’s boundaries. They contact the SAP in order to communicate with internal entities.

  – **Internal Entities**
    * are all components located inside the system’s boundaries.

  – **Single Access Point**
    * provides an interface that allows external parties to communicate with system internal components.
    * gathers information about the occurring access requests, their origin and authorization information.
    * triggers actions or forwards data to parties inside the system.

• **Collaborations**

  The SAP class interacts with any class that needs to communicate with system external parties. It works as a mediator. If certain security policies need to be enforced, all requests might be sent to a Check Point class before they are forwarded to their intended recipients.

• **Behavior**

  The first UML sequence-diagram in Figure 4 shows how an external request is routed to an internal component by the Single Access Point. If the recipient is unknown or not available the SAP can be implemented to send an error message back to the calling instance. This sequence of events is depicted in Figure 5. The following examples relate to the online store class diagram in Figure 2. Figure 6 shows an outside Customer placing an order and Figure 7 depicts an example of a request that cannot be forwarded by the SAP.

\(^7\) compare *Check Point Security Pattern* in section 4.2 at page 27
Figure 4: UML sequence-diagram: Single Access Point: Recipient available

Figure 5: UML sequence-diagram: Single Access Point: Recipient not available

Figure 6: UML sequence-diagram: Single Access Point: Example 01
Figure 7: UML sequence-diagram: Single Access Point: Example 02

- **Constraints**

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticity</td>
<td>A message that is directly sent to an internal component originates either from a system internal component or the SAP.</td>
</tr>
<tr>
<td>Availability</td>
<td>If the internal recipient to an external request is not available an error message is sent to the calling external entity.</td>
</tr>
<tr>
<td>Availability</td>
<td>If external entities send more requests to the SAP than usual their forwarding can be delayed or skipped in order to maintain availability of service for other entities.</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Communication between internal components is not disclosed to outside entities.</td>
</tr>
<tr>
<td>Integrity</td>
<td>Messages inside the system or from the SAP to an internal component cannot be modified by external entities.</td>
</tr>
<tr>
<td>Internal Existence</td>
<td>If the internal recipient to an external request is known it is forwarded.</td>
</tr>
<tr>
<td>Internal Existence</td>
<td>If the internal recipient to an external request is unknown an error message is sent to the calling external entity.</td>
</tr>
</tbody>
</table>
• Consequences

<table>
<thead>
<tr>
<th>Accountability:</th>
<th>The accountability is improved by application of this pattern. Single Access Point provides a good place to capture logging information and perform authentication tasks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality:</td>
<td>Confidentiality can also be improved as disclosure of information to unauthorized parties is more unlikely. Every access will pass the Single Access Point and can be monitored.</td>
</tr>
<tr>
<td>Integrity:</td>
<td>Undesirable modification of data can be prevented better by efficient checks who is allowed to access the system.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Availability may be reduced if the Single Access Point cannot handle all accesses concurrently. Denial of Service (DoS) Attacks can be prevented more efficiently. All information that is necessary for their detection can be gathered at the Single Access Point.</td>
</tr>
<tr>
<td>Performance:</td>
<td>If substantial checking and logging tasks are conducted at the Single Access Point, system performance can be affected. Multiple Access Points would provide more flexible and easier ways to communicate.</td>
</tr>
<tr>
<td>Cost:</td>
<td>Depending on the extent of communication with external parties, the development can be more difficult and expensive.</td>
</tr>
<tr>
<td>Manageability:</td>
<td>Maintenance of security-relevant code will be easier as it is not scattered over the entire system.</td>
</tr>
<tr>
<td>Usability:</td>
<td>In order to access the system, a user might have to enter more information than the authentication of the specific request would require. Since the interface, provided by the SAP, has to conduct all possible communication with external entities, it may be unified. This unification may lead to a more inconvenient login process.</td>
</tr>
</tbody>
</table>

• Implementation

To be determined.

• Known Uses
– UNIX telnet application
– Windows NT login application

• Related Security Patterns
The Check Point Security Pattern enforces the current security policy, by monitoring the traffic passing the Single Access Point. Once a request is verified positively, the Session pattern can be used to grant an external party access rights that exceed the current request. The Role Security Pattern facilitates the management of access rights when assigned to a Session.

• Related Design Patterns
Singleton, Facade, Wrappers

• Supported Principles
The main idea of the Single Access Point pattern is the principle keep it simple (Principle 6). Security-relevant code for the examination of system accesses is not any longer distributed over the whole system. If interfaces to external components are considered potential weaknesses, then the pattern also secures weak links (Principle 1). Furthermore, the implementation of a central monitoring instance indicates a certain reluctance to trust incoming messages and queries (Principle 9).

• Community Resources
To be determined.
4.2 Check Point

The Check Point pattern that is explained herein is also part of Yoder’s and Barcalow’s framework for application security [16]. The Parts Structure, Participants, Collaborations, Constraints, Consequences and Supported Principles contain new information. The other sections were based in part on Yoder’s and Barcalow’s pattern.

- **Pattern Name and Classification**
  The Check Point pattern is a structural pattern.

- **Intent**
  Check Point proposes a structure that checks incoming requests. In case of violations the Check Point is responsible for taking appropriate countermeasures.

- **Also Known As**
  The Check Point pattern is also referred to as [16]
  - Access Verification,
  - Authentication and Authorization,
  - Holding off hackers,
  - Validation and Penalization or
  - Make the punishment fit the crime.

- **Motivation**
  In order to prevent unauthorized access it is vital to check who interacts in which manner with a system. It can be a difficult task to determine whether a given access should be granted or not. Any secure system needs a component that monitors the current communication and takes measures if necessary. Furthermore, usability should not be affected by denying innocuous actions. These tasks are performed in a specifically designated component. It is called a Check Point.

- **Applicability**
  Check Points are applicable in any security-relevant communication. It can be used at each abstraction-level from inside-application-level to network-level.
In order to perform a check, the system needs to have a policy that will be enforced. The Check Point implementing that policy should be able to distinguish between user mistakes and malicious attacks.

- **Structure**

A checkpoint is a component that analyzes requests and messages. In order to monitor message flow it needs to be placed between different parties that exchange data. A good position to install a Check Point is at a place where it can intercept a major part of the traffic. Hence, a Single Access Point is predestinated to be combined with a Check Point - all messages will be monitored. This structure is depicted in Figure 8.

![Figure 8: UML class-diagram: Check Point](image-url)
• **Participants**

– Check Point

* implements a method to check messages according to the current security policy.
* triggers actions that might be necessary to protect the system against attackers.
* grants messages access, if they are considered innocuous.

– Countermeasure

* provides a set of actions that can be triggered in order to react to a access violation.

– Security Policy

* object that implements the rules that are applied to determine whether an access or condition is allowed or not.

• **Collaborations**

The Check Point class monitors the message flow between other components. The calling class requests the Check Point to scrutinize a certain message. To determine whether a request is entitled or not, the *Security Policy* object is consulted. Once it is approved, according to the policy, the request will be forwarded to its intended recipient. If illegal request attempts or attacks are uncovered then the Check Point triggers appropriate counteractions (implemented in a *Countermeasure* object).

• **Behavior**

The UML sequence-diagrams in figure 9 and 10 show two possible scenarios. The first one depicts the message flow, if access is granted to an external request. The second one shows a message being rejected by the Check Point. Concrete examples of message interaction are shown in Figures 11 and 12.
Figure 9: UML sequence-diagram: Check Point: access granted

Figure 10: UML sequence-diagram: Check Point: access denied
Figure 11: UML sequence-diagram: Check Point: Example 01

Figure 12: UML sequence-diagram: Check Point: Example 02
Figure 13: UML state-diagram: Check Point
- **Constraints**

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticity</td>
<td>Check Point’s Requests concerning the current Security Policy are only answered by the authentic Security Policy Object.</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>The communication between Check Point and the Security Policy may not be overheard by external entities.</td>
</tr>
<tr>
<td>General Security</td>
<td>If a checked message is consistent to the implemented security policy, the request may be forwarded to its intended recipient.</td>
</tr>
<tr>
<td>General Security</td>
<td>If a checked message is potentially harmful and further action is required according to the implemented security policy, the appropriate actions are triggered.</td>
</tr>
<tr>
<td>General Security</td>
<td>If a checked message is potentially harmful according to the implemented security policy, the message is not forwarded to its intended recipient.</td>
</tr>
<tr>
<td>Integrity</td>
<td>Messages sent between Check Point and Security Policy cannot be modified.</td>
</tr>
</tbody>
</table>
• Consequences

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability:</td>
<td>is not affected</td>
</tr>
<tr>
<td>Confidentiality:</td>
<td>The application of this pattern can contribute to the confidentiality of a system if the check algorithm is good (i.e. it detects and stops unauthorized use).</td>
</tr>
<tr>
<td>Integrity:</td>
<td>Undesirable modification can be filtered if the check algorithm is capable of detecting those attacks.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Denial of Service (DoS) attacks can be prevented, if the Check Point algorithm takes appropriate actions (e.g. delaying or ignoring messages if they match a certain pattern).</td>
</tr>
<tr>
<td>Performance:</td>
<td>Complex check routines may slow down the system and its message exchange.</td>
</tr>
<tr>
<td>Cost:</td>
<td>The implementation of a security-policy is difficult and therefore costly</td>
</tr>
<tr>
<td>Manageability:</td>
<td>Maintenance of security-relevant code will be easier if it is located at one position. Though, complexity of the check algorithm is - depending on the implemented policy - high.</td>
</tr>
<tr>
<td>Usability:</td>
<td>Some communication activity might be prevented even if it is not harmful. A high-quality check algorithm is vital.</td>
</tr>
</tbody>
</table>

• Implementation

To be determined.

• Known Uses

– UNIX telnet application
– Windows NT login application

• Related Security Patterns

The Check Point Security Pattern enforces the current security policy, by monitoring the traffic passing the Single Access Point. Once a request is verified positively the Session pattern can be used to grant an external party access rights that exceed the current request.
The *Role Security Pattern* facilitates the management of access rights when assigned to a Session.

- **Related Design Patterns**
  The Strategy Design Pattern [7] can be used to decouple Check Point from the actual implementation of the security policy. Changes can be adopted more easily.

- **Supported Principles**
  The Check Point pattern can be used to *secure the weakest link* (Principle 1). Applied in combination with other security mechanisms it *practices defense in depth* (Principle 2). If the *principle of least privilege* (Principle 4) is part of the current security policy it will be implemented in the check algorithm, used by Check Point. Monitoring messages and other data shows *reluctance to trust* that they are harmless (Principle 9).

- **Community Resources**
  To be determined.
4.3 Roles

The following Security Pattern is part of Yoder’s and Barcalow’s [16] framework of architectural patterns for application security. Furthermore it corresponds with Fernandez’ Role-Based-Access-Control (RBAC) [5] pattern. We present the Roles pattern in our for the security domain tailored template. Therefore it was enriched with behavioral and structural information, UML diagrams and a Constraints section.

• Pattern Name and Classification
The following pattern is termed Roles. In accordance to the classification approach chosen by Gamma et al. [7], it belongs to the class of structural patterns as it proposes a infrastructure that is based on the composition of classes and objects.

• Intent
The Roles pattern aims for better maintainability of privileges in a system. Thus, it follows Principle 6 (Keep it simple) of the Ten Principles by abstracting rights from specific subjects and thereby facilitating maintenance of complex privilege structures.

• Also Known As
In accordance to Yoder and Barcalow [16] the Roles pattern is also known as
   - Actors,
   - Groups,
   - Projects,
   - Profiles,
   - Jobs and
   - User Types.

• Motivation
Various multiuser-systems provide functionality or resources that may not be used by every user. In order to control access to those restricted parts the system has to keep track of the privileges a user has. Commonly privileges are not derived from an individual’s needs.

---

8The mentioned Security Pattern Template is outlined in Section 3.2.
but from the tasks that a subject is supposed to perform in the system. Thus, abstracting privileges from persons and assigning them to profiles termed *Roles* can be useful. Given some business application (e.g. Enterprise Resource Planning (ERP) Software), certain tasks and queries relate to a specific job positions. For example, the right of employee A to query all corporate cost centers is derived from his job position as Manager of Finance not from the person itself. If A leaves the company all his rights need to be revoked and transferred to B the new Manager of Finance, who will now perform A’s tasks. Unfortunately A had a great amount of privileges and the administrative effort caused by A’s cancellation is enormous. In some constellations with individuals having more than one position and quitting just a part of them, it is even harder to determine and maintain privileges.

The application of the *Roles* pattern simplifies maintenance in such scenarios a lot. As all privileges needed to perform a certain task are assigned to its respective role, it is sufficient to match users with roles according to their tasks. This approach is by far more efficient while dealing with complex privilege structures than the plain assignment of rights to users. Once all needed roles are created and rights are assigned to them, they are usually not anymore subject to frequent changes.

The *Roles* pattern belongs to the field of *Security Patterns*, because it addresses problems that threaten system security. Its main contribution is the simplification of the security-critical administration process of privileges.

- **Applicability**

  The *Roles* pattern is applicable to any system that needs to restrict access of participating subjects (i.e. users or processes) to its resources. Especially in large systems that have many users with similar privilege constellations, the implementation of *Roles* can reduce organizational redundancy.

  In systems where it is hard to identify common privilege groups and nearly every user needs a role of his own the implementation of this pattern generates no additional benefit. Single-user systems or systems that have static privileges structures are not subject to the application of the *Roles* pattern at all.

- **Structure**

  The *Roles* pattern is realized by constructing a User-Role-Privilege relationship. Instead of
assigning privileges to users, Role objects are created. These objects contain a set of privileges. Several Role objects can be assigned to each user. According to the Principle of Least Privilege (Principle 4 of the Ten Principles), a user has no privileges unless they are granted to him by an assigned Role object. The different Role objects may be aggregated in a collection object (in our example termed: Roles). This structure is reflected in the UML class diagram in Figure 14. We used the previous example of an online store to show how the Roles pattern works in combination with the other Security Patterns presented earlier. Those dependencies are subject to further elucidation in Section Related Security Patterns of this pattern.

**Note:** The information that connects users and privileges is stored inside the Role object. We made this design decision in order to keep security code together. An implementation in which all privileges are checked for certain roles upon a call, would be slow and scatter security code throughout the system [16].

- **Participants**
  - Privilege
    * defines activities or resource use that is permissable to subjects that where granted this privilege.
    * if violations should be checked automatically, the definition of the privilege needs to be formal.
  - Role
    * holds a set of privileges that are associated to that specific role object.
    * saves the users to which this role is assigned.
    * provides information about its users and privileges.
  - Roles
    * is a collection of all roles in the system.
  - User
    * stores information about an user.
    * exists for each user of the system.
Figure 14: UML class-diagram: Roles
• **Collaborations**
  Role objects contain links to Privilege and User objects. As the User-Role-Privilege relationship determines the rights of certain users it is critical to maintain consistency. Thus, upon deletion of User or Privilege objects, the affected Role objects have to be updated. Ideally this would be done by some notification mechanism between the

• **Behavior**
  Not applicable.

• **Constraints**
  To be determined.
### Consequences

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability</td>
<td>is not affected</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>The Roles pattern can contribute to confidentiality by improving the administration of access rights.</td>
</tr>
<tr>
<td>Integrity</td>
<td>Integrity is affected in a similar way as Confidentiality. The likelihood of unintentionally granting privileges that harm system’s integrity is reduced</td>
</tr>
<tr>
<td>Availability</td>
<td>Restriction of resources is used to maintain availability. The Roles pattern also simplifies the administration of privileges that deal with resource management.</td>
</tr>
<tr>
<td>Performance</td>
<td>A great amount of Role objects can slow down the process of checking whether a subject has the needed privileges or not. Given an efficient implementation and assuming reasonable use of Roles, overall performance can be improved.</td>
</tr>
<tr>
<td>Cost</td>
<td>Maintenance cost can be reduced in complex multiuser systems. Development is more difficult and more expensive on account of the higher level of complexity. Privilege lookups are more complicated and may take longer.</td>
</tr>
<tr>
<td>Manageability</td>
<td>The administrator’s possibilities of managing privileges are enhanced, as User-Role and Role-Privilege relationships can be managed separately. This enables an administrator to make use of the additional level of indirection to group common privileges [16]. If numerous privilege combinations exist, a significant amount of roles is necessary. Manageability will be improved if the overall amount of relationships is reduced by the application of Roles.</td>
</tr>
<tr>
<td>Usability</td>
<td>In complex systems a user might perceive slightly longer delays in case of extensive privilege checks.</td>
</tr>
</tbody>
</table>

### Implementation

To be determined.
• **Known Uses**
  To be determined.

• **Related Security Patterns**
  The *Roles* pattern can be combined with several other *Security Patterns*. A *Check Point* can use the *Roles* to determine whether operations may be permitted or not. *Session* objects can provide links to the *Roles* that were assigned to the current user. The privilege structure of the *Roles* pattern can be used by the *Limited View* pattern to define what is visible to the user.

• **Related Design Patterns**
  An elegant way of maintaining consistency in the User-Role-Privilege structure is to use the *Observer* [7] pattern. In this way, Role objects will be notified upon deletion of either Privilege or User objects and can take appropriate actions to restore consistency.

  If the system needs to behave differently, depending on the current user’s Role, Yoder and Barcalow [16] propose the use of the *Strategy* [7] pattern.

• **Supported Principles**
  Given appropriate preconditions⁹, the *Roles* pattern simplifies the administration and helps to keep track of the privileges. Hence, it embodies Principle 6: *Keep it simple*. Furthermore, the organization and enforcement of privileges should follow Principle 4 (*Principle of least privilege*) and grant a subject only those rights it subject is explicitly authorized for.

• **Community Resources**
  To be determined.

---

⁹outlined in Section *Applicability* of this pattern
4.4 Session

Alike the other so far presented patterns, the *Session Security Pattern* belongs to Yoder’s and Barcalow’s [16] framework of architectural patterns for application security. Below we transferred it to our security-specific template and enriched it with additional structural and behavioral information. The *Session* pattern presented herein may also be used for non-security purposes. We focus in our presentation only on its utilization for security improvement.

**• Pattern Name and Classification**

The *Session* pattern deals with object creation (of *Session* objects), accordingly we classify it as creational pattern.

**• Intent**

The intention of the *Session Security Pattern* is to provide different parts of a system with global information about a user, who currently interacts with internal components. With a focus on security this is important, because it may be used to facilitate accountability and the enforcement of privilege violations.

**• Also Known As** Other names used to address the *Session* pattern are

- User’s Environment,
- Namespace,
- Threaded-based Singleton and
- Localized Globals.

**• Motivation**

In many systems global information is needed at various positions. Especially security relevant code in multiuser systems needs information on the current users. Usually the *Singleton* pattern would be used to instantiate one global object that holds and provides all required data. Unfortunately, this approach is not applicable in a distributed, multi-user or multi-threaded environment. Several instances might be required, for example one for each user or process.

To overcome this problem *Session* objects are used. They provide the needed information. In terms of security, *Session* objects are very useful to keep security-relevant information
like roles, privileges or authentication data. This information enables components inside the system to conduct further checks based on the provided information (Principle 2: *Practice defense in depth*). A *Session* object can be created after an user has successfully logged into the system. Typically, this would be done at the *Check Point*\(^{10}\) of a system. In this way, the system can identify a specific user to be authorized for certain operations by checking the respective *Session* object.

- **Applicability**
  
  The *Session* approach can be used if different parts of a system need access to global information that cannot be provided by a *Singleton*. Furthermore, *Session* objects are not shared by different instances of an application that run concurrently or different users.

- **Structure**

  A *Session* object is created during runtime and shares global data with other system components. In order to facilitate access to a *Session* object from various positions in the system, a mechanism for distributing its instance needs to be implemented. Figure 15 depicts how the *Session* pattern could be integrated in our example system.

- **Participants**

  - *Session*
    *
    * stores information that will be provided by *Session*.
    *
    * the stored information is initialized on creation.
    *
    * provides methods for sharing the information.
    
  - System components that use *Session*
    *
    * need to know the instance of the *Session* object they use.
    *
    * call methods of session to retrieve information.

- **Collaborations**

  System internal components that require the global information need to have the instance of their *Session* object. Depending on the design of the system, they might be given the

\(^{10}\)see Section 4.2 for more information
Figure 15: UML class-diagram: Session
respective Session instance with a user’s request to perform some action. Once they have access to the instance, it can be used to query the desired information.

- **Behavior**

  Figure 16 depicts the message flow upon an user’s access request. After the Check Point receives a message from an user it demands the corresponding privilege information from a Session object. In the example we outlined here, the access request is permissable according to the privilege structure and the message is forwarded to its intended recipient.

  ![UML sequence-diagram: Session: Example](image)

  Figure 16: UML sequence-diagram: Session: Example

- **Constraints**

  To be determined.
- **Consequences**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accountability:</strong></td>
<td>Using a <em>Session</em> object a user’s actions can be traced and logged.</td>
</tr>
<tr>
<td><strong>Confidentiality:</strong></td>
<td>By providing role and privilege information in a user’s <em>Session</em> object, unauthorized access can be recognized and stopped by internal components.</td>
</tr>
<tr>
<td><strong>Integrity:</strong></td>
<td>In a similar way as confidentiality, integrity of an user’s operations can be checked against his privilege structure in order to identify actions that compromise integrity.</td>
</tr>
<tr>
<td><strong>Availability:</strong></td>
<td>Alike Confidentiality and Integrity, Availability can be improved by applying the <em>Session</em> pattern. Resource restrictions might be checked according to an user’s privileges.</td>
</tr>
<tr>
<td><strong>Performance:</strong></td>
<td>Depends on implementation.</td>
</tr>
<tr>
<td><strong>Cost:</strong></td>
<td>Development can be difficult. All system components that need global information need to have an instance of <em>Session</em>. With increasing amount of information stored in a <em>Session</em> object more organization is necessary. Those circumstances may influence development cost.</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>Manageability, especially during system development is enhanced by the <em>Session</em> pattern. The pattern establishes a common mechanism to handle important global variables. Adding a variable to the <em>Session</em> object will make it available to all components. Instead of exchanging various variables only the instance of <em>Session</em> needs to be transferred. [16]</td>
</tr>
<tr>
<td><strong>Usability:</strong></td>
<td>Not affected.</td>
</tr>
</tbody>
</table>

- **Implementation**
  
  To be determined.

- **Known Uses**
  
  To be determined.
• Related Security Patterns
The Session pattern can be easily integrated with the other Security Patterns. As mentioned above, the Check Point provides a convenient place to create a instance of Session, once a user is authenticated successfully. Furthermore, the Session can keep the current users Roles and make them accessible to system internal components. Finally, the Limited View pattern can query the Session object in order to determine which data the current user sees.

• Related Design Patterns
The from its goals the Singleton [7] is closely related to the Session pattern, which takes into account multi-threaded, multi-user or distributed environments [16].

• Supported Principles
The Session can facilitate defense in depth (Principle 2), by providing system components with security-relevant information. Furthermore, Principle 6 (Keep it simple) is supported by this pattern: Access to global variables and the passing of global data is simplified and thus less error-prone. It is crucial to keep in mind that privacy should be promoted (Principle 7): The Session object should not provide security-critical information (like passwords) to system components that leak and compromise this information.

• Community Resources
To be determined.
4.5 Full View With Errors

Yoder and Barcalow [16] introduced the Full View with Errors Security Pattern in their work. The following section adapts the pattern to our security-tailored template.

- **Pattern Name and Classification**
  
  The Full View with Errors pattern is a behavioral pattern. It describes how various parts of the system that deal with user interaction could behave.

- **Intent**
  
  The Full View with Errors pattern depicts an approach to prevented users from performing illegal operations by failing them securely (Principle 3).

- **Also Known As** The pattern Full View with Errors is also known as
  
  - Full View with Exceptions,
  - Reveal All and Handle Exceptions and
  - Notified View.

- **Motivation**
  
  Systems user interaction need to concern about how to prevent reliably that a user cannot perform restricted operations. Furthermore, the designers need to notify the user if he tries to run actions that are not permissable.

  The Full View with Errors pattern proposes to show a user all options, no matter whether his current Roles or privileges allow him to choose all of them. If an illegal operation is performed the system will notify the user and explain why the chosen action is not permitted.

- **Applicability**
  
  The Full View with Errors pattern is applicable to systems that have user interaction. Furthermore, it is not reasonable to use Full View with Errors if there are many operations a user is not able to perform. In such cases where many meaningless options exist the user is distracted from the main functionality (intended for that specific user).
• **Structure**
  Not applicable.

• **Participants**
  Not applicable.

• **Collaborations**
  Not applicable.

• **Behavior**
  The *Full View with Errors* pattern displays all available options. Upon execution of an operation a check is performed that determines whether the action is legal or not. In case of an illegal call the operation is stopped and an error message is displayed.

• **Constraints**
  To be determined.
• Consequences

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Confidentiality:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Integrity:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Not Affected.</td>
</tr>
<tr>
<td>Performance:</td>
<td>On the one hand the approach of the Full View with Errors pattern does not require checks before a view is updated. Hence, it is usually faster as the Limited View pattern. On the other hand the user may choose illegal operations that are subject to costly checking.</td>
</tr>
<tr>
<td>Cost:</td>
<td>Cost is not significantly affected by the application of the Full View with Errors pattern.</td>
</tr>
<tr>
<td>Manageability:</td>
<td>Implementation of this pattern is fairly easy as one has not to concern about different views.</td>
</tr>
<tr>
<td>Usability:</td>
<td>By displaying all options - no matter whether they are available at the moment or not - the user does not get confused by missing items. Unfortunately, the user might be tempted to choose an illegal operation and will evoke an error message. Frequent security notifications can annoy the user.</td>
</tr>
</tbody>
</table>

• Implementation

To be determined.

• Known Uses

To be determined.

• Related Security Patterns

The Full View with Errors pattern competes with the Limited View pattern. They have a opposing strategy. The error-check upon a call of an operation usually makes use of the structure provided by the Roles pattern. This structure can be accessed via a Session object.

• Related Design Patterns

Not applicable.
• Supported Principles
  The main principle underlying the Full View with Errors pattern is Fail Securely (Principle 3). Furthermore, development can be simplified by applying this pattern. Thus, Principle 6: Keep it simple is also subject to this pattern.

• Community Resources
  To be determined.
4.6 Limited View

The Limited View pattern belongs to Yoder’s and Barcalow’s [16] framework. In the following section it is presented in terms of our Security Pattern template.

• Pattern Name and Classification
  The Limited View is a behavioral pattern, as it proposes a certain behavior for visualization components of a system.

• Intent
  The Limited View intends to prevented users from performing illegal operations by offering only valid operations to them. According to Principle 2 (Practice defense in depth) a Limited View can be an additional security measure.

• Also Known As
  Other known names for the Limited View pattern are
  – Blinders,
  – Child Proving,
  – Invisible Road Blocks,
  – Hiding the Cookie Jars and
  – Early Authorization

• Motivation
  Users should be deterred from performing restricted operations. The Limited View pattern prevents a user to choose illegal actions by not visualizing those options.
  In order to archive this goal, the view has to be customized according to the current users access privileges. This implies that a user’s privileges are checked in advance. Thus, the Limited View pattern may access via a Session object.

• Applicability
  Alike Full View with Errors, the Limited View pattern is applicable to systems that interact with users. A visualization based on the patterns concept of hiding irrelevant information is
reasonable if there are many options a user is not allowed to access anyway. In some cases the 
Limited View pattern may be applied in order to concisely display the available functionality.

- **Structure**
  Not applicable.

- **Participants**
  Not applicable.

- **Collaborations**
  Not applicable.

- **Behavior**
  The Limited View pattern only displays options that the current user is entitled to access. 
  Thus, a check is performed in advance to the visualization of the different options. Hence, 
  any operation that the user may chooses is consistent with his privileges.

- **Constraints**
  To be determined.
• **Consequences**

<table>
<thead>
<tr>
<th>Accountability:</th>
<th>Not affected.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Integrity:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Not Affected.</td>
</tr>
<tr>
<td>Performance:</td>
<td>In advance to the visualization each option is checked for consistency with the current user’s rights. Those tasks can derogate system’s performance. Once, all options are displayed some systems do not conduct further checks to improve performance.</td>
</tr>
<tr>
<td>Cost:</td>
<td>Cost is not significantly affected by the application of the Limited View pattern.</td>
</tr>
<tr>
<td>Manageability:</td>
<td>The implementation of this pattern may be more challenging, as the complete right structure needs to be checked simultaneous. However, a combined security check could be more efficient and simplified.</td>
</tr>
<tr>
<td>Usability:</td>
<td>The changing options in the interface can confuse the user if the meaning of appearing and disappearing options is not evident to him. It is less likely that an user sees error screens and gets frustrated by them.</td>
</tr>
</tbody>
</table>

• **Implementation**

To be determined.

• **Known Uses**

To be determined.

• **Related Security Patterns**

The *Limited View* pattern embodies the converse approach to the *Full View with Errors* pattern. The structure needed to preselect the options that qualify for a specific user, might be provided by a *Session* object. To map privileges and users the *Roles* pattern could be used.

• **Related Design Patterns**

Yoder and Barcalow [16] propose to make use of the *State, Strategy, Composite* and *Builder*
Design Patterns during the implementation of the Limited View pattern.

• Supported Principles
The Limited View pattern embodies several of Viega’s and McGraw’s [14] Ten Principles. Principle 2 (Practice defense in depth) can be accomplished if the preselection of what the user can do is used additionally to other security checks. According to Principle 4 (Principle of least privilege) and Principle 7 (Promote privacy) this pattern restricts the user to acquire more information than necessary. This is also consistent with the idea of Principle 8, which states that hiding a secret is hard and a secure system has to consider insider attacks. Hence, the Limited View pattern realizes Principle 9 (Reluctance to trust). Along with applying this pattern, the system can be simplified by combining security checks and meets Principle 6 (Keep it simple).

• Community Resources
To be determined.
4.7 Authorization

The Authorization pattern was presented in Fernandez' work *A pattern language for security models* [5]. It is a Security Pattern of high abstraction that has been used in various systems. We present it here in order to show the use of our security-specific template.

- **Pattern Name and Classification**
  
  The Authorization pattern is a structural Security Pattern.

- **Intent**
  
  Authorization provides a structure that facilitates access control of resources.

- **Also Known As**
  
  Not applicable.

- **Motivation**
  
  Many systems need to restrict access to its resources according to certain criteria (e.g. a security policy). To structure the different possibilities of access, we distinguish between active entities and passive resources [5]. Below, active entities are also referred to as subjects, who access passive resources (protection objects) of a system. A uniform way that abstracts from the type of subjects and protection objects is highly desirable. The Authorization pattern takes account of these goals and provides an abstract structure that is suitable for representing access conditions in a computational environment. Furthermore, the concept presented in the Authorization pattern offers features to facilitate and model transferral of rights as well as a restriction of the conditions in which authorization rules apply.

- **Applicability**
  
  The Authorization pattern is applicable in any system that requires supervision of subjects, who access resources.

- **Structure**
  
  The authorization structure of a system can be captured in classes and relationships among them. Active entities are represented by instances of a Subject class and and passive resources by instances of a Protection Object class. Between those main participants exists a relation...
that portrays which subject is entitled to access certain objects. The properties of this relationship are organized in the association class *Rights*. The objects of this class define the type of access, transferal conditions and applicability of the respective *Authorization* relation between *Subject* and *Protection Object*.

![UML class diagram](image)

*Figure 17: UML class-diagram: Authorization (In accordance to Fernandez [5])*

- **Participants**
  - **Protection Object**
    * represents passive resources of the system that are accessed by subjects.
  - **Rights**
    * defines the properties of the authorization rule between subject and protection object.
    * The *access_type* property describes which kind of access of the current *Rights* object grants. Commonly this property holds values in the style of read, write, execute, ...
    * The *constraint* property is a predicate that describes under which circumstances the current *Rights* object is valid and may grant a certain privilege.
    * The *transferable* property determines whether a right is transitive and its connected subject may grant the right to other active entities.
  - **Subject**
    * represents active entities that need to access protected objects in accordance to their rights.

- **Collaborations** Different *Subjects* in the system want to access *Protection Objects*. In order to make use of a certain resource they need to request access to it from the responsible
controlling instance. This instance will check if a association class between the subject and the object exists that justifies the required access request. Depending on this examination access is granted or not.

- **Behavior** The behavior of the Authorization pattern is depicted in Figure 18 and Figure 19. The first diagram shows a subject that is successfully authorized using the herein presented structure. The second sequence shows how the request of an active entity is rejected.

![UML sequence-diagram: Authorization approved](image)

**Figure 18: UML sequence-diagram: Authorization approved**

- **Constraints**
  
  To be determined.
• Consequences

<table>
<thead>
<tr>
<th>Accountability</th>
<th>Not affected.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>Can be improved by specification and rigorous enforcement of rights.</td>
</tr>
<tr>
<td>Integrity</td>
<td>Can be improved by specification and rigorous enforcement of rights.</td>
</tr>
<tr>
<td>Availability</td>
<td>Can be improved by specification and rigorous enforcement of rights.</td>
</tr>
<tr>
<td>Performance</td>
<td>The system’s performance might be derogated by extensive right checks and the evaluation of the rights’ constraint predicates.</td>
</tr>
<tr>
<td>Cost</td>
<td>Cost is not significantly affected by the application of this pattern.</td>
</tr>
<tr>
<td>Manageability</td>
<td>Thanks to the high level of abstraction, the Authorization pattern may be applied to manage any type of subject (processes, users, roles, groups, ...) resource (transactions, files, devices, ...) and access (read, write, execute, ...) [5].</td>
</tr>
<tr>
<td>Usability</td>
<td>If the access rights are checked extensively the user might recognize a loss of performance.</td>
</tr>
</tbody>
</table>

• Implementation

To be determined.
• **Known Uses**
  To be determined.

• **Related Security Patterns**
  The *Check Point* pattern can be used to examine requests by using the structure of the *Authorization* pattern. The application of the *Session* approach can provide the system with the subject-privilege information. The *RBAC* or *Roles* pattern is an extended version of this pattern.

• **Related Design Patterns**
  Not available.

• **Supported Principles**
  The *Authorization* pattern should be used in combination with Principle 4 (*Principle of least privilege*) to assign the user’s rights. Furthermore, *Authorization* can be used to *compartmentalize* (Principle 5) the system and reduce the impact of a security breach. For example, an attacker that illegally manages to authenticate as user A, has only the rights user A would have. Given a restrictive security policy, the enforcement of access rights *promotes privacy* (Principle 7).

• **Community Resources**
  To be determined.
4.8 Multilevel Security

The Multilevel Security pattern is part of Fernandez’ Pattern language for security models [5]. It belongs to the high abstraction level Security Pattern and is of special interest for the development of high-security systems (like military applications). The following section adapts this pattern to our Security Pattern Template.

- Pattern Name and Classification
  The Multilevel Security pattern is a structural pattern, as it establishes a structure for systems that partition subjects and objects in different security levels.

- Intent
  This pattern tries to provide a mechanism for handling access in a system with various security classification levels.

- Also Known As
  Not applicable.

- Motivation
  In many systems integrity and confidentiality of data need to be guaranteed. For example in the domain of military intelligence, the classification of data and the clearance of users need to be considered to accomplish these goals.

  The Multilevel Security Security Pattern addresses these concerns and provides a structure that allows to have different security levels for subjects and objects. Furthermore, the presented approach facilitates the enforcement of access restrictions in a environment of the mentioned characteristics.

- Applicability
  The Multilevel Security pattern is applicable to systems that need to provide several security levels for subjects and objects. Furthermore, a hierarchical structure that reflects the subjects and objects sensitivity level has to exist. According to Fernandez [5] this precondition is violated in most commercial environments.

- Structure
  To represent the Multilevel Security structure for each subject exists an instance of the Sub-
ject Classification class and for each object an instance of the Object Classification class. These instances are used to aggregate a subject’s respective an object’s security levels and categories. Given the domain of military intelligence, examples of security levels could be: top secret, secret, confidential and unclassified. Categories are predefined compartments that represent object’s matter respective the subject’s clearance to that compartment.

Figure 20: UML class-diagram: Multilevel Security (In accordance to Fernandez [5])

- **Participants**
  - Object Category
    * defines a category that a object belongs to.
  - Object Classification
    * determines the security classification of an object (passive resource that is accessed by subjects).
    * The object’s security classification is represented by a set of object categories and object levels.
  - Object Level
    * defines the security level of an object.
  - Subject Category
    * defines a category a subject has access to.
  - Subject Classification
* determines the security classification of an subject (active entity that accesses objects).

* The subject’s security classification is represented by a set of object categories and object levels.

- **Subject Level**
  * defines the security level of the subject.

**Collaborations**

The classes *Subject Classification* and *Object Classification* contain a set of category and level classes. This set determines the security classification of the respective object. Access will be granted if the classification of the requesting subject dominates the classification of the protected object. The *Behavior* section explains in detail how to determine whether one classification dominates a second one.

**Behavior**

The UML sequence diagrams in Figure 21 and Figure 22 depict the message flow upon a subject’s access request in a multilevel security system. The first scenario shows a successful authorization. The second scenario portrays the a subject’s request being rejected. The decision on whether to grant or to deny access is based on the participating subject’s and object’s security classification. Below we describe the underlying rules for evaluation of access. Access is granted if a subject’s classification $C_s$ dominates an object’s classification $C_o$. To preserve confidentiality the rules of the Bell-LaPadula [3] model are applied. Bell’s and LaPadula’s *simple security* property (also known as *no read up* property) of the model prevents subjects without appropriate clearance to access higher classified data. According to Bell-LaPadula, the different security levels are ordered between a highest and lowest level. An exemplary order might be:

unclassified $<$ confidential $<$ secret $<$ top secret

where *unclassified* states the lowest possible sensitivity level and *top secret* the highest. Given the ordered level structure, $C_s$ dominates $C_o$ if the classification level of $C_s$ is greater than the classification level of $C_o$ and $C_o$’s categories are a subset of $C_s$’s clearance categories. Integrity is addressed by similar rules that are stated in Biba’s model, which is outlined in Summer’s book on *Secure*
Computing [12]. Biba defines rules that prevent subjects of high integrity levels to observe data of lower integrity in order to prevent corrupted data of being taken to higher integrity levels. Furthermore, it is not permissable for subjects to modify objects of higher integrity levels.

Figure 21: UML sequence-diagram: Multilevel Security: Access granted

- **Constraints**
  To be determined.
Figure 22: UML sequence-diagram: Multilevel Security: Access denied

- **Consequences**

<table>
<thead>
<tr>
<th>Accountability</th>
<th>Not affected.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality:</td>
<td>User access is controlled according to the rules that are stated in the Bell-LaPadula [3] model in order to guarantee confidentiality of data.</td>
</tr>
<tr>
<td>Integrity:</td>
<td>Biba’s model [12] and its rules establish integrity.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Not affected.</td>
</tr>
<tr>
<td>Performance:</td>
<td>The evaluation of access rights can derogate performance.</td>
</tr>
<tr>
<td>Cost:</td>
<td>Establishing a multilevel security system can be costly. All subjects and objects need to be classified in certain sensitivity levels.</td>
</tr>
<tr>
<td>Manageability:</td>
<td>The Multilevel Security pattern can ease administrative work in a environment that requires classification of subjects and objects.</td>
</tr>
<tr>
<td>Usability:</td>
<td>The users range of action might be limited by restrictive rules.</td>
</tr>
</tbody>
</table>

- **Implementation**
  To be determined.

- **Known Uses**
• **Related Security Patterns**
  The *Check Point* pattern may be used to enforce the *Multilevel Security* structure, which can be provided in *Session* object.

• **Related Design Patterns**
  Not available.

• **Supported Principles**
  The *Multilevel Security* pattern and its mechanism of granting access to subjects implements the *Principle of least privilege* (Principle 4). Furthermore, it facilitates *Compartmentalization* (Principle 5) and reduces the impact of a security breach. *Promoting privacy* (Principle 7), *Hiding secrets is hard* (Principle 8) and *Reluctance to trust* (Principle 9) are main ideas of the *Multilevel Security* pattern.

• **Community Resources**
  To be determined.
References


