INFORMATICA, 2010, Vol. 21, No. 2, 229–246
© 2010 *Institute of Mathematics and Informatics, Vilnius*

Extended Software Architecture Based on Security Patterns

Dušan SAVIĆ, Dejan SIMIĆ, Siniša VLAJIĆ
Faculty of Organizational Sciences, University in Belgrade
Jove Ilica 154, 11000 Belgrade, Serbia
e-mail: {dules, dsimic, vlajic}@fon.rs

Received: September 2008; accepted: May 2009

Abstract. One of the major activities in software design is defining software architecture. Before designing software structure and software behavior we have to define its architecture. In this paper we have proposed three-tiered software architecture. This software architecture extends application logic tier with security. We have implemented two important security issues: authentication and authorization processes. These processes are implemented through software patterns. The software patterns have the particular place in the Proposed Software Architecture (PSA). In this paper, we have presented these software patterns and explained why they are important in PSA.

Keywords: software architecture, software patterns, authentication and authorization processes, software development process.

1. Introduction

In application development, especially the enterprise ones, some resources must be available only to a certain number of users. Such resourses must be protected. Therefore, there is a need to ensure access to protected resources only to authorized users (Tseng *et al.*, 2008).

Identification, authentication, authorization and access control are central to computer security (Choo, 2006). Identification states that client is the user is and authentication proves the identification claim. The identification can be proved by: applying a password which only the user knows, digital certificate (a key, usually kept in some of the folders), smart card injection in a special reader or biometric measuring such as finger prints. Once authenticated, access control rules decide what the user is authorized to do and what he is not.

Before the user approaches protected resource, the user directly approaches some of the services that execute authentication. These services use specific authentication mechanisms (e.g., Single-Sign-on) and implement some of the authentication models. Some of these modes are explained in the Anderson (2003) works:

- plain-text password,
- encrypted password,
- password forwarding,
- X509 certificates.

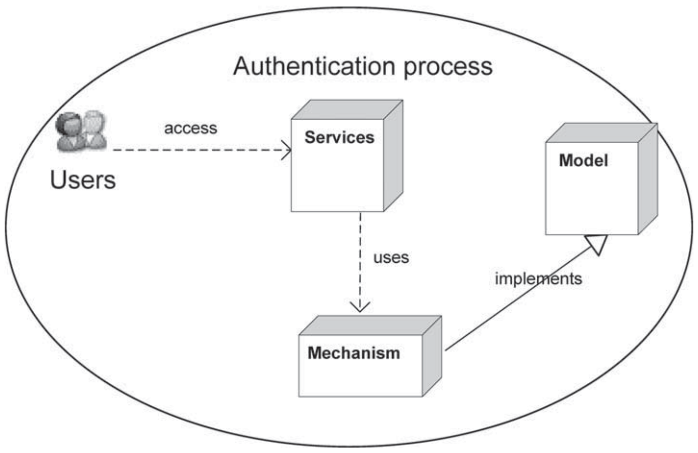

Fig. 1. General model of authentication process.

The Fig. 1 shows general model of authentication process.

A good part of application security design is about deciding what technologies to use for identification and authentication, how to specify access control, and how to manage authorizations (Kumar, 2003). In software development two major security issues are most important: authentication process and authorization process. As emphasized in the previous definition, application security design needs to give answers to two key questions. The first one is what technologies to use for identification and authentication and the second one is how to manage authorizations. We have started with the hypothesis that we can describe authentication and authorization processes with software patterns and that we can propose software architecture which will support different security solutions for authentication and authorization processes. Design software patterns are idependent of the programming languages so, they stand as general design practices for common classes of software problems including security. Also, we have assumed that we can use design patterns to describe these processes because they provide a way to solve issues related to software development using proved solution.

In this paper we have described Proposed Software Architecture (PSA). PSA presents an extended version of the software architecture that Larman presented in the book *Applying UML and Patterns* (Larman, 2001). Larman has not considered security problems. Therefore, the main aim of this paper is to extend this software architecture with the security component. This security component implements two major security issues: authentication and authorization process. We have implemented security component through software pattern, so the authentication and the authorization processes are independent of any security solutions (SpringSecurity; JavaSESec) which implement authentication and authorization models. The aim of this paper is to propose software architecture that can support different security solution. This software architecture can also support user implementation of the authentication and authorization models (for example, Plain-text password) in a particular way.

This paper covers topics such as software architecture, security and software patterns. Therefore, a context of this paper is software architecture (Larman's software architecture; Larman, 2001) and security, while software patterns are part of the solution. The problem we have considered is how to extend Larman's software architecture with security. The solution is Proposed Software Architecure.

In the Section 2 we have presented short overview on software patterns. Section 3 contains some definitions of the software architecture, metamodel of the software architecture that we suggested and short overview software architecture that Larman (2001) proposed. The following section describes PSA. PSA contains security component. We have explained the role that security component has in PSA. The PSA is presented through software patterns so in the next section we have described software patterns and explained the importance that they have in PSA. The special place in implementation security processes is given to the GoF design software patterns (Gamma *et al.*, 1995). At the end of this paper we have presented an example of application for candidates enrollment on the faculty.

2. Software Patterns

Pattern originated as an architecture concept by Christopher Alexander (1979) but gained popularity in computer science after the book *Design Patterns*: *Elements of Reusable Object-Oriented Software* was published in Gamma *et al.* (1995). Alexander tells:

> *Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing* (Alexander, 1979).

As emphasized in the previous definition, each pattern is a three-part rule: problem and solution in a certain context. According to the Alexander's definition of the pattern, we can present the aim of this paper as a pattern. The context of this paper is software architecture and security. The problem is in two important security issues (authentication and authorization processes) that not are considered in Larman's software architecture. The solution is PSA which extends Larman's software architecture with security component. This security component implements authentication and authorization processes through software patterns. The following Fig. 2 presents problem, solution and context of this paper. Coplien says:

> *Patterns usually describe software abstractions used by advanced designers and programmers in their software. As abstractions, patterns commonly cut a cross other common software abstractions like procedures and objects, or combine more common abstractions in powerful ways* (Coplien, 2000).

In this paper we have used GoF design patterns (Gamma *et al.*, 1995) to define the software architecture and to describe authentication and authorization processes. We have used the following design patterns to define the software architecture: *Facade, Template method, Bridg* and *Command*. Also, we have used *Chain of Responsibility*, *Singleton*, *Factory method* and *Strategy* to describe and implement authentication and authorization

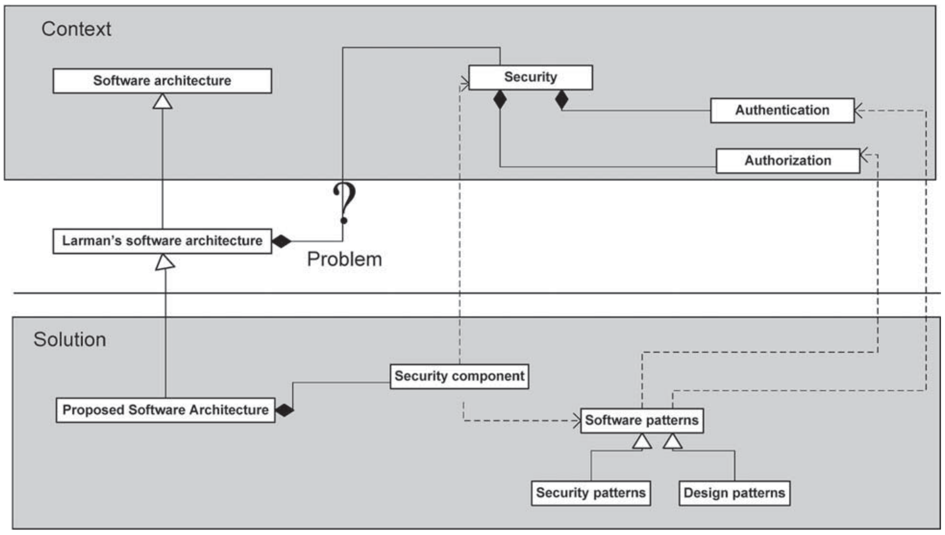

Fig. 2. The problem, solution and context of the paper.

processes. More details about these patterns and role that they have in PSA are presented in the Section 5.

3. Software Architecture and Security

One definition of *software architecture* is:

> *An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition* (Booch *et al.*, 1998).

According to this definition, is the definition provided by Bass *et al.* (2003):

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise: software components, the externally visible properties of those components and their relation among them.*

On one hand software architecture must abstract some information from the system but, on the other hand, it must present enough information to make the software system understandable (Heiberg, Matskin and Pedersen, 2002; Bajec and Vavpotic, 2008). If there is no abstraction, then there is no architecture. Software architecture shows and describes structure, behavior and constraint of the software. The Fig. 3 shows our metamodel of the software architecture. This metamodel is in accordance with the recommendations that are proposed in *Guide to the Software Engineering Body of Knowledge* (Abran *et al.*, 2005).

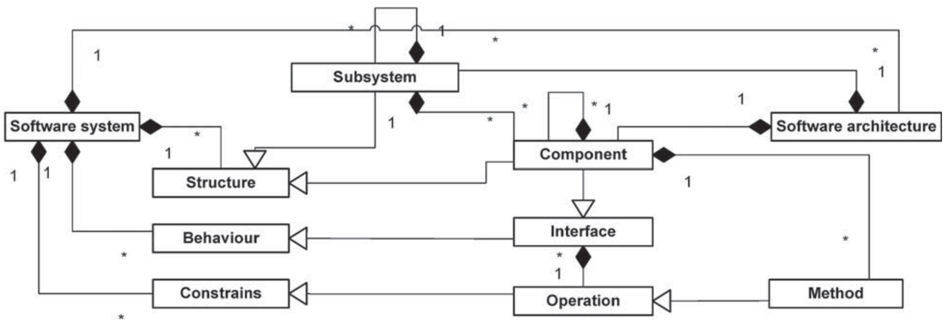

Fig. 3. Software architecture metamodel.

The software behavior is described through intefaces and operations. Each operation is abstraction that contains the name of the operation, parameters and return value. Software components contain the methods that implement operations of the interface. The operations on the interface are public so they are visible to the users.

In his book *Applying UML and Patterns*, Larman (2001) explains three-tiered software architecture. That software architecture we call *Larman's Software Architecture* (LSA). He emphasizes that one common software includes presentation tier, aplication logic tier and storage tier. Also Larman divides the application logic tier into two layers: domain layer and services layer. The first one is used to define the structure of a system, and the other one is for defining its behaviour. He introduces the concept of partitions and mentions that layers of an architecture represents the vertical tiers, while partitions represent a horizontal division of relatively parallel subsystems of a layer. These partitions present functional entities responsible for reporting, security, business logic or communication. The Fig. 4 shows all parts of this LSA architecture.

In this paper we have used a *Simplified LSA* (SLSA). In SLSA we have paid special attention to the service layer. Within this service layer we have created business logic partition. These partition is implemented through the following software components: *Controller*, *Business logic* and *Broker*. The Fig. 5 shows this SLSA architecture.

Controller component is responsible to accept the request from the client, forward that request to the *Bussines logic* component and turn back response to the client. *Bussines logic* component is responsible to perform client request. It accepts request from the *Controller* component, performs it and turns back the response to the *Controller* component while *Broker* component presents one of the possibilities to implement persistent framework. This framework is responsible for the materialization, dematerialization and caching objects in the operating memory. It is established on the Hollywood's principle: "*Don't call us, we'll call you*" which means that the user-defined classes accept messages only from the predefined classes of the persistent framework (Larman, 2001). This concept is also known as Dependency Injection (Fowler, 2004) and callback concept (Meyers, 2001). Therefore, the main idea of this paper is to extend these SLSA with the security component. This security component implements two major security issues: authentication and authorization process. In the next section we have proposed software architecture

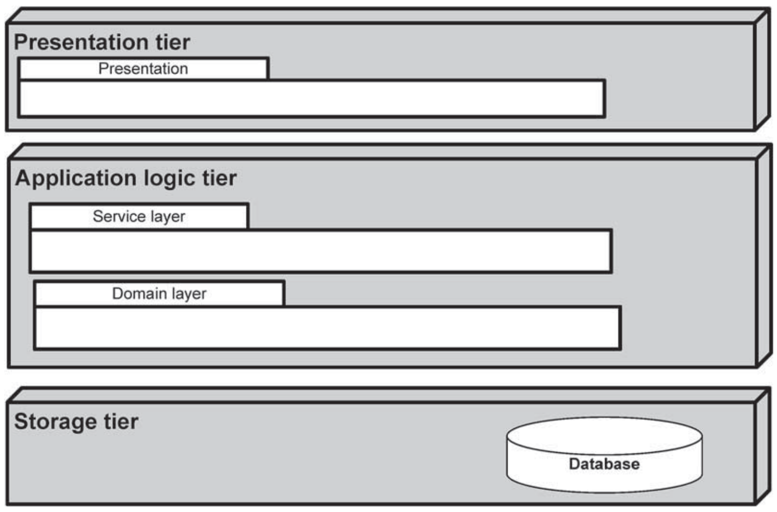

Fig. 4. Larman’s Software Architecute (LSA).

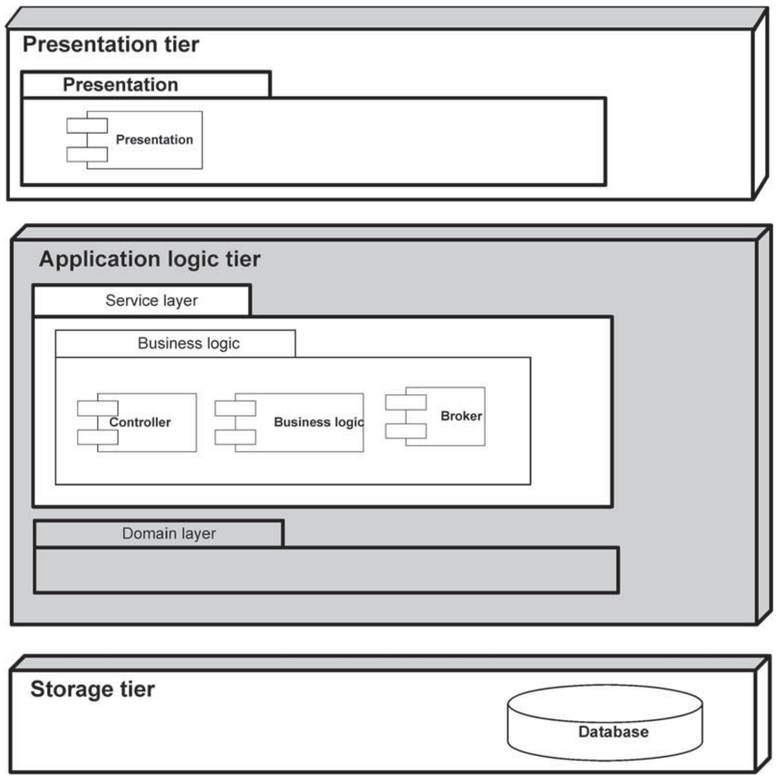

Fig. 5. Simplified Larman’s Software Architecute (SLSA).

that contains this security component. We have implemented security component through software pattern, so the authentication and the authorization processes are independent of any security solutions that implement authentication and authorization models. The aim of this paper is to propose software architecture that can support different security solutions that implement different authentication and authorization models. This software achitecure can also support user's implementation of the authentication and authorization models (for example, Plain-text password) in a particular way.

The whole Java platform gives a solid basis for writing reliable and secure Java applications. Beside standard libraries there are additional ones, that can be applied to solve some specific problems in security domain (JavaSESec). Some of these libraries are listed below:

- Java Cryptography Architecture (JCA),
- Java Cryptographic Extension (JCE),
- Java Certification Path API (CertPath),
- Java Secure Socket Extension (JSSE),
- Java Authentication and Authorization Service (JAAS),
- Java Generic Secure Services (JGSS).

Security problem in the enterprise application development (Hatebur *et al.*, 2007; Hafiz *et al.*, 2007; Meland and Jensen, 2008) is also supported by certain specifications such as Java Platform Enterprise Edition (JEE) servlet specification (JavaServlet] and Enterprise JavaBeans (EJB) specification (EJB] which are surely some of the most important ones. In enterprise Java applications development Spring security gives a complete security solution (SpringSecurity].

4. Proposed Software Architecture

As we have described in Section 3, Larman emphasizes that application logic tier consists of a domain and services layers pointing out that a service layer consists of several partitions, for example, a report and security partitions. In the same section we have described SLSA. We have extended service layer of the SLSA with security partition. This security partition contains security component. The Fig. 6 shows application logic tier with security partition.

The security in PSA covers two important issues: the authentication and the authorization processes. This processes are implemented through several activities. We have distinguished activities such as: creating the security context, verifying user's authentication, verifying user's authorization, calling the execution of the system operation. For each of these activities we created a class responsible for its execution. These classes are low coupled between them, on one hand, and with high cohesion on the other hand. This is very important because our authentication process remains independent even if we change the authentication model. So, we have described authentication and authorization processes as a chain which consists of nodes, where each node represents an activity. In order to successfully execute the authentication and/or authorization processes, we need

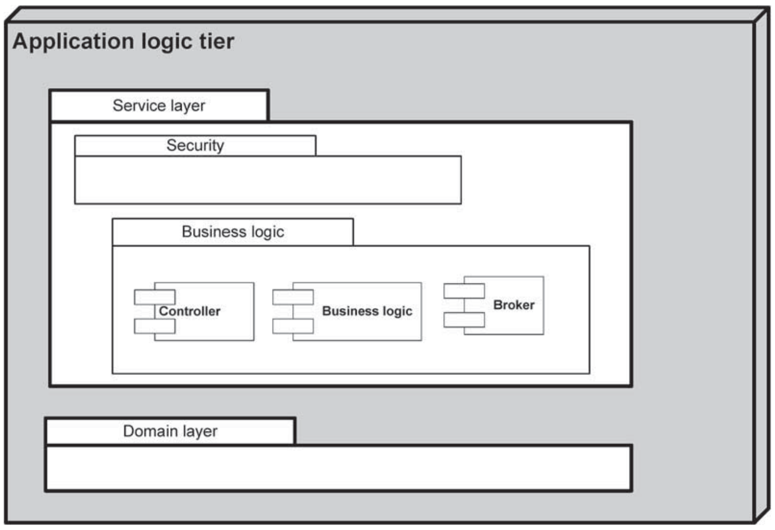

Fig. 6. Extended Larman's software architecute.

to execute all the activities of these processes. We have used GoF Chain of Responsibility design pattern to implement this security chain. The Fig. 7 shows the classes that create this security chain.

Class *ContextNode* is the first one in the security chain and it is responsible for creating *security context*. The security context is presented as a container which contains some important protection attributes such as user data, user group, and access level.

Class *AuthenticationNode* is responsible for user authentication. This class executes the user's authentication. As we have emphasized in the Section 1 there are several authentication models and *AuthenticationNode* class is responsible to choose one of them to implement authentication process. As the result of authentication process an object is created which represents the authenticated user (object of the *Auth*[1] class) and is saved into security context so the other classes in the chain could read a relevant data that this object contains. The main difference between our security component and Spring security solution is in this object.

Class *SecurityChainExceptionNode* is responsible for generating an error in case that the authentication or authorization processes fail.

Class *AuthorizationNode* is responsible for user's authorization. For the authenticated user whose data are in the security context, *AuthorizationNode* class is responsible to verify if the user can perform action (system operation) that he wants. If the authenticated user is authorized to perform the system operation, the last class in the chain *CallSONode* will be called.

Class *CallSONode* is the last in the chain and it is responsible to call a specific system operation. For each system operation there is a class responsible for its execution (GRASP

[1]Class *Auth* is described in Section 5.

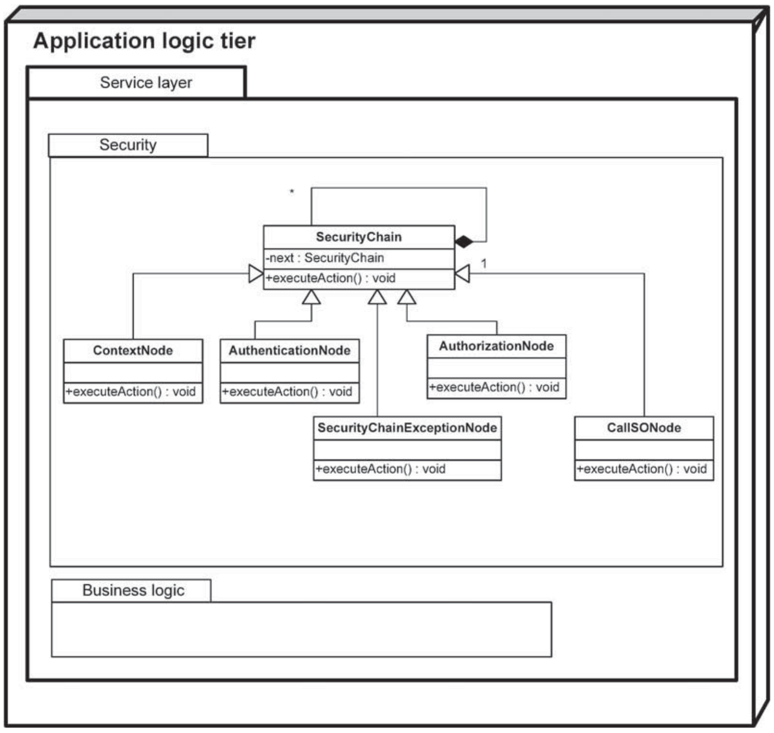

Fig. 7. Class diagram of security chain.

pattern of the high cohesivity) (Larman, 2001). Class *CallSONode* calls the particular class responsible for performing some bussines logic dynamically based on its name. The Fig. 8 shows PSA.

In PSA, the controller component (*Controller*) accepts the request to perform operation from the client. It forwards this request to the security component which determines whether the user is authenticated and authorized to perform operation. The security component through security chain verifyes if the user is authenticated and authorized to perform operation. If the user data are in the security context and if *AuthenticationNode* class authorizes the user to perform the requested operation, the last class in the secutity chain (*CallSONode*) calls the class responsible for performing the operation (*SaveSO*).

Software patterns are used to implement all components of the software architecture. Access to the service layer is enabled through the implementation of the *Single Access Point* pattern (Schumacher *et al.*, 2006). We have used the GoF *Façade* structure pattern (*Controller* component) to implement *Single Access Point* pattern.

The following section describes the patterns we have used in implementation security component.

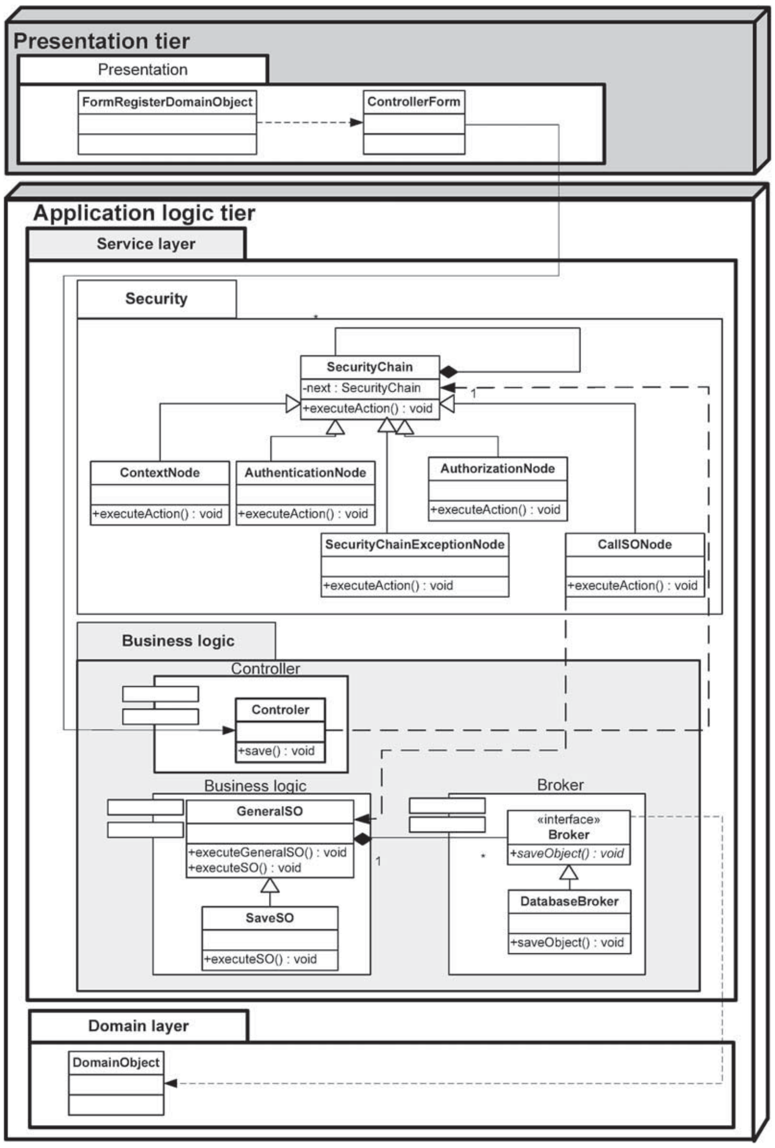

Fig. 8. Proposed Software Architecture (PSA).

5. Applying Patterns in Authentication and Authorization Processes

In authentication and authorization processes of *PSA Auth* class holds the central position. The relation of this class to the other classes is shown in Fig. 9.

This class presents a changed structure of ROLE-BASED ACCESS CONTROL pattern (Steel *et al.*, 2005; Schumacher *et al.*, 2006). Class *SOperation* presents a system operation and corresponds to a *ProtectionDomain* class of the ROLE-BASED ACCESS

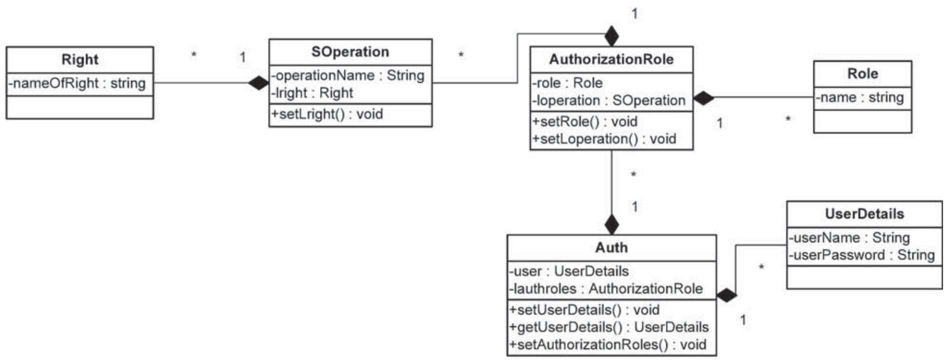

Fig. 9. The sructure of a changed ROLE-BASED ACCESS CONTROL pattern.

CONTROL pattern (Muhammad *et al.*, 2008). Several user rights (*Right*) can be related to one system operation (*SOperation*), while each system operation can be performed only by the user (*UserDetails and Auth*) with exactly corresponding role (*AuthorizationRole*).

The authentication and authorization processes are implemented through the GoF behaving pattern *Chain of Responsibility*. The chain consists of several classes and each class is responsible to perform specific activities such as: creating the security context, verifying user authentication, verifying user authorization, calling the execution of the system operation.

GoF *Singleton* pattern is used to create the security context activity. The security context is presented by *SecurityContext* class and contains an *Auth* class. This class saves the information about the authenticated user and his rights. Class *SecurityContextCreator* has only one instance and it is implemented as a *Singleton* pattern. Class diagram in Fig. 10 shows the classes that take part in the creating security context activity.

Activities such as verifying user authentication and verifying user authorization are represented by the Check Point security pattern (Steel *et al.*, 2005; Schumacher *et al.*, 2006). Also, we have used the GoF pattern: *Strategy* and *Factory Method* to implement verifying user authentication activity and the *Factory Method* pattern to implement verifying user authorization activity. Class diagram in Fig. 11 shows the classes that take part in verifying user authentication activity, while Fig. 12 shows the classes that take part in verifying user authorization activity.

As described in the introduction, authentication process preforms services that use specific mechanisms of authentication and implement some of the authentication models. We have created classes that are responsible to perform different strategies of the authentication proces via GoF Strategy design pattern.

Class *AuthorizationNode* performs verifying user authorization activity. This class via *AuthorizationManager* class, which creates the corresponding *AuthorizationProvider* class (depending on the type of the storage where data for determining user right for accessing system operation are kept, the appropriate *AuthorizationProvider* class is created) is responsible for determining user's right to access the system operation. For an authenticated user whose data are in the security context, verification is performed to find out

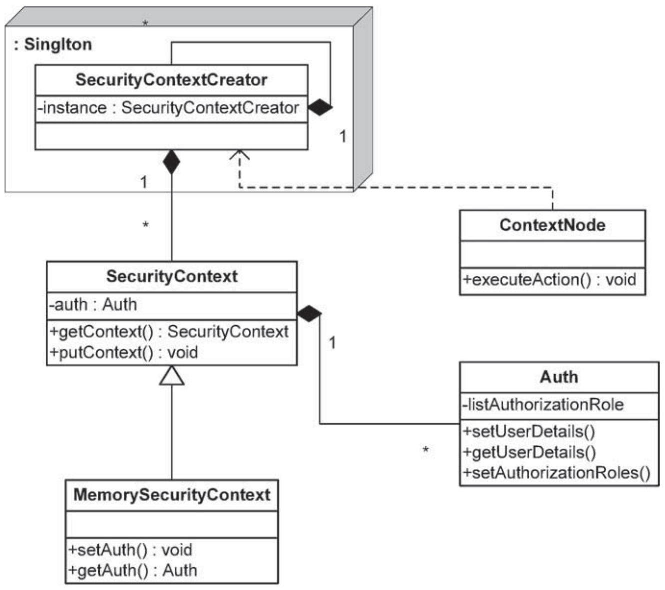

Fig. 10. The classes taking part in security context creating activity.

if the required system operation can be performed. The method *isAuthorized* of the *AuthorizationProvider* class executes the user authorization. If the user is authenticated and authorized to perform a specific system operation, then the last class in the chain (*CallSONode*) will be called. The class *CallSONode* calls the class responsible for performing the requested system operation.

Class *CallSONode* calls a specific class, that is responsible to perform a requested system operation, dynamically based on its name. We have used GoF *Command* pattern to implement this activity. The class that represent the requested system operation, is implemented as a *Receiver* class. Class diagram in Fig. 13 shows the classes that take part in this activity. The Fig. 14 shows software patterns we have used in PSA.

6. Case Study

The system under development is Candidates Enrollment at the Faculty System (CEFS). We have a brief description of the system we have developed. Candidates submit a request for enrollment at the faculty (submitted the required documents), take a entry test, and enroll in the faculty. Administrative workers accept the required documents from the candidates and register them into system. The commissions put the entries of the test results into database. It is necessary to enter each test three times in order to reduce the possibility of errors. The central commission corrects the errors that make commissions to enter the test results. The central commission prepare rank list of candidates.

In this brief description of CEFS we can see that user can be administrative workers, member of commission to enter the test results or/and member of central commission.

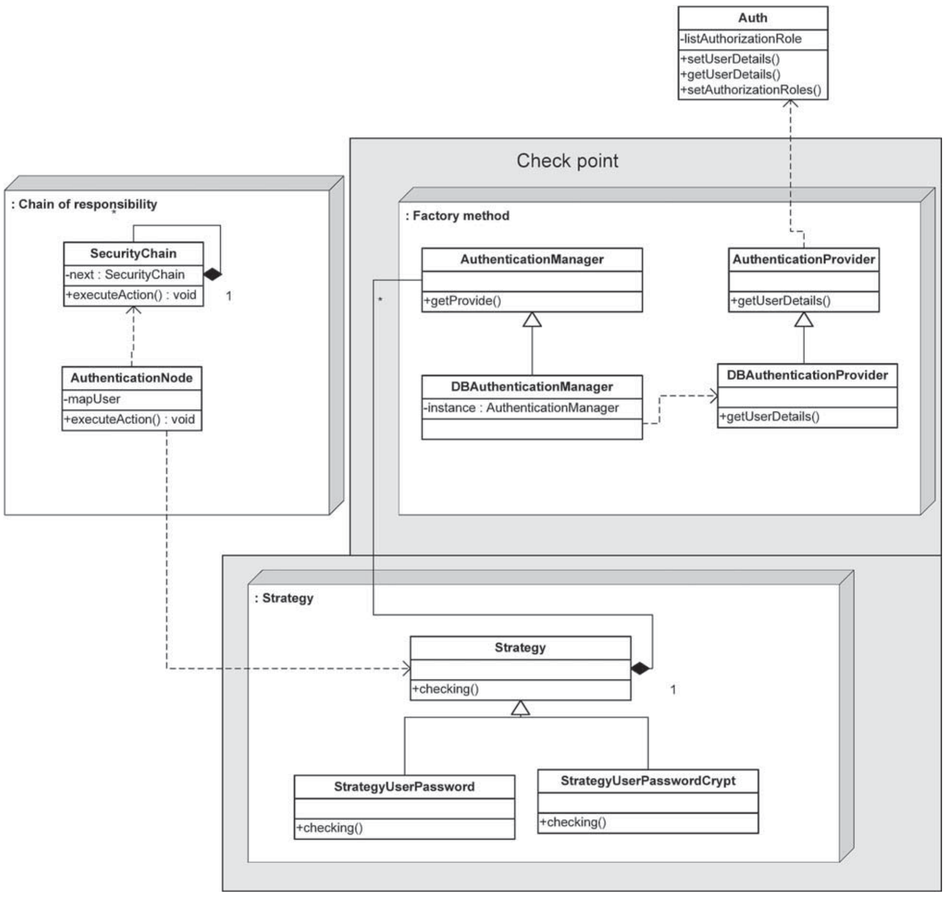

Fig. 11. The classes taking part in the authorization process activity.

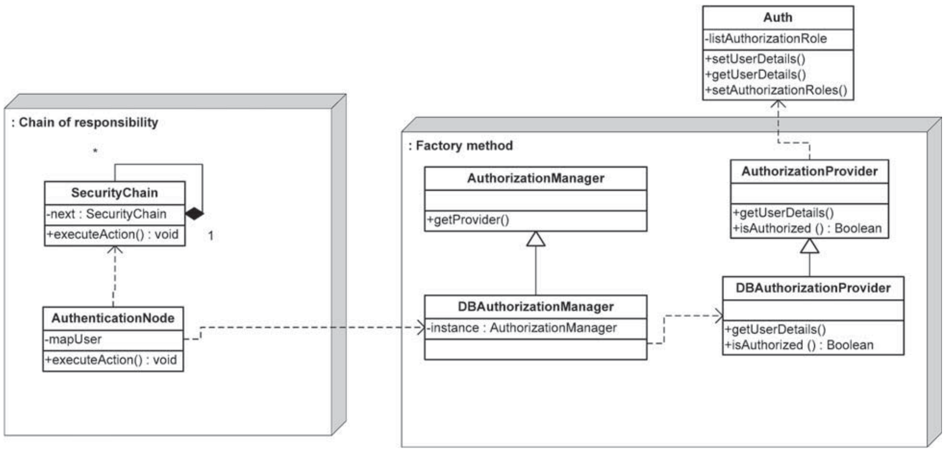

Fig. 12. The classes which take part in the authorization process activity.

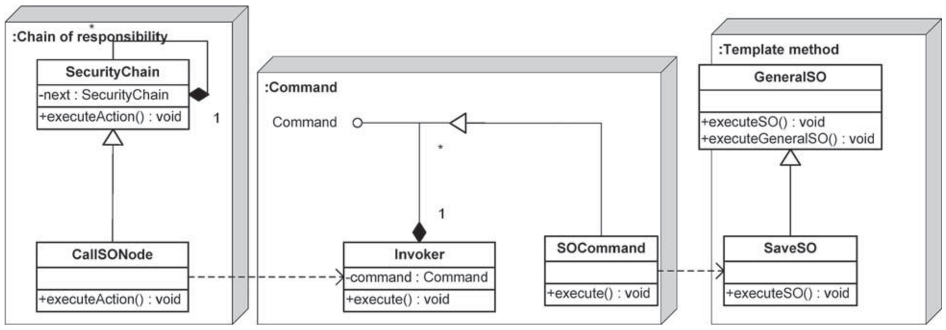

Fig. 13. The casses taking part in system operation calling activity.

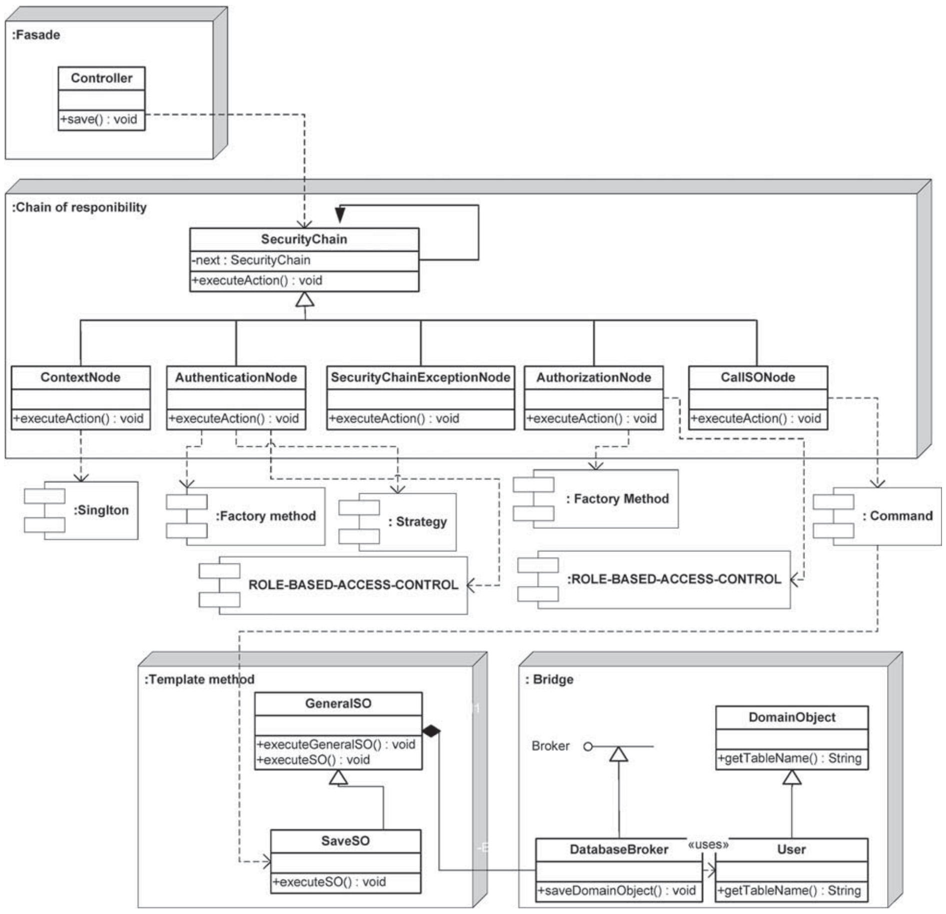

Fig. 14. Software patterns used in the PSA.

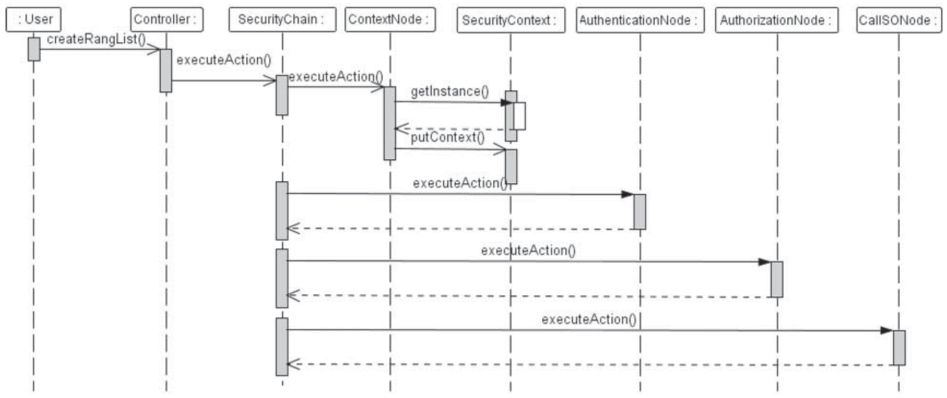

Fig. 15. UML sequence diagram of createRangList() system operation.

Use case Create Rank List can perform only the user who is the member of central commission. The UML sequence diagram (Fig. 15) describes execution of the operation that create rang list.

The user performs operation createRangList() on the Controller. Controller accepts request and forward it to the SecurityChain. The executeAction() method of the ContextNode class will be performed the first. This method creates security context and calls executeAction() method the following class in the chain. The executeAction() method of the AuthenticationNode class via AuthenticationManager class create AuthenticationProvider class. This class is responsible to save user data in the security context. The method getAuthenticatedUser() of the SecurityContext class returns object (Auth) that represents authenticated user. Method isAuthorized() of the AuthorizationProvider class is responsible to determine is the user authorized to perform this operation. If the user is authorized to perform createRangList()operation the last class in the chain will be called. The method executeAction() of this class (CallSONode), calls system operation CreateRangList().

7. Conclusion

In application development, especially the enterprise ones, some resources must be available only to a certain number of users. Such resourses must be protected. Therefore, there is a need to ensure access to protected resources only to authorized users. Accordingly, an application security design has to give answers to two key questions. The first one is what technologies to use for identification and authentication and the second one is how to manage authorizations. Also, in software development process, the main artifact in software design is its architecture. These topics, software architecture and security, are covered in this paper.

The aim of this paper we are presented as a pattern. The context of this paper is software architecture and security. The problem covers two important security issues (authentication and authorization processes) that are not considered in Larman's software

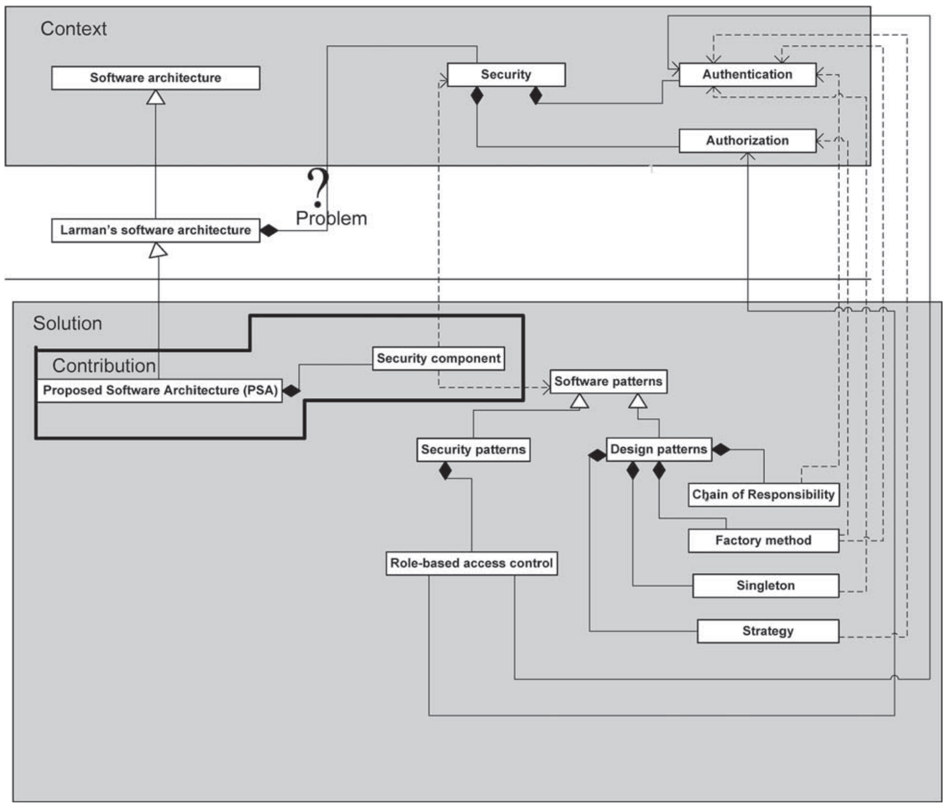

Fig. 16. The aim and solution of the paper.

architecture. The solution is PSA which extends Larman's software architecture with security component. This security component implements authentication and authorization processes throught software patterns. The Fig. 16 shows the aim and solution of this paper.

At the end we can conclude, that we can extend Larman's software architecture with security component. So, the main contribution of this paper is PSA. Also, we have confirmed the assumption that we can use design pattern to describe and implement authentication and authorization processes independent of any security solutions. We have used the following design patterns: *Chain of Responsibility* to describe authentication and authorization processes, *Strategy* to support different authentication models, *Singleton* to create security context, *Factory Method* to support creating objects without specifying the exact class of the object that will be created in authentication and authorization processes, and security pattern *Role-based access control* which presents authenticated and authorized user.

Also, in this paper we have presented a general model of authentication process and metamodel of the software architecture.

References

Abran, A., Moore, J., Bourque, P., Dupuis, R., Tripp, L. (2005). *Guide to the Software Engineering Body of Knowledge – 2004 Version – SWEBOK*. IEEE-Computer Society Press, Institute of Electrical and Electronics Engineers. `http://www.swebok.org/htmlformat.html`

Alexander, C. (1979), *The Timeless Way of Building*. Oxford University Press, New York.

Anderson, P. (2003). *Authentication Models Explained*: *A Background Tosingle-sign-on Issues for the University of Edinburgh*. School of Informatics University of Edinburgh, University of Edinburgh. `http://www.ucs.ed.ac.uk/ucsinfo/cttees/citc/work/authdirwg/explain.pdf`

Bajevac, M., Vapotic, D. (2008). A framework and tool-support for reengineering software development methods. *Informatica*, 19(3), 321–344.

Bass, L., Clements, P., Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley.

Booch, G., Rumbaugh, J., Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

Choo, K.-K.R. (2006). On the security analysis of Lee, Hwang & Lee (2004) and Song & Kim (2000) key exchange/agreement protocols. *Informatica*, 17(4), 467–480.

Coplien, J.O. (2000). Software patterns. In: *Bell Labs*. The Hillside Group, pp. 7–12.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison Wesley/Longman Inc.

Fowler, M. (2004). *Dependency Injection*. `http://martinfowler.com/articles/injection.html`

Hafiz, M., Adamczyk, P., Johnson, R. (2007). Organizing security patterns. In: *IEEE Software*, pp. 52–60.

Hatebur, D., Heisel, M., Schmidt, H. (2007). A security engineering process based on patterns. In: *Database and Expert Systems Applications, DEXA '07*: *18th International Conference*. Regensburg, pp. 734–738.

Heiberg, T., Matskin, M., Pedersen, J. (2002). An agent-based architecture for customer services management and product search. *Informatica*, 13(4), 441–454.

Java SE security. `http://java.sun.com/javase/6/docs/technotes/guides/security/index.html` (accessed on July, 2008).

JSR-000154 Java^{TM} `http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html`

JSR 220:Enterprise JavaBeans^{TM} `http://jcp.org/aboutJava/communityprocess/edr/jsr220/`

Larman, C. (2001). *Applying UML and Patterns*: *An Introduction to Object-Oriented Analysis and Design*, 2nd ed. Prentice Hall.

Kumar, P. (2003). *J2EE Security for Servlets, EJBs and Web Services*: *Applying Theory and Standards to Practice*. Prentice Hall PTR.

Mayers, S. (2001). *Effective STL*: *50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley Professional Computing Series. Addison-Wesley.

Meland, P., Jensen, J. (2008). Security software design in practice. In: *2008 Third International Conference on Availability, Reliability and Security*, pp. 1164–1171.

Muhammad, A., Hafner, M., Breu, R. (2008). Constraint based role based access control in the SECTET-frameworkA model-driven approach. *Journal of Computer Security*, 16(2), 223–260.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P. (2006). *Security Patterns, Integration Security and System Engineering*. Wiley.

Spring security. `http://www.acegisecurity.org/guide/springsecurity.html` (accessed on July, 2008).

Steel, C., Nagappan, R., Lai, R. (2005). *Core Security Patterns*: *Best Practices and Strategies for J2EE^{TM}, Web Services, and Identity Management*. Prentice Hall.

Tseng, Y., Wu, T., Wu, J. (2008). A pairing-based user authentication scheme for wireless clients with smart cards. *Informatica*, 19(2), 285–302.

D. Savić received the BSc degree in information system and technologies from the Faculty of Organization Sciences, University of Belgrade, in 2004. He is currently postgraduate student and teaching assistant on Faculty of Organizational Sciences at the Department of Information System and Technologies. His main research interests include: software development process, software design, software pattern, meta-modeling and software requirements engineering.

D. Simić, PhD, is an associate professor at the Faculty of Organizational Sciences, University of Belgrade. He received the BS in electrical engineering and the MS and the PhD degrees in computer science from the University of Belgrade. His main research interests include: applied information technologies and security of computer systems and networks.

S. Vlajić, PhD, is an assistant professor of software engineering at University of Belgrade – Faculty of Organizational Sciences – Department of Information Systems and Technologies – Software Engineering Laboratory, in Belgrade, Serbia. He has taught undergraduate and graduate level courses: introduction to programming, introduction to information system, software design, software patterns, programming methodology and Java programming language. He wrote many books, scripts and publications about C++, Java, software design, software patterns, database and information systems. His main research interests include: software process, software design, software maintenance, software pattern formalisation and programming methodology.

Tipiniais apsaugos projektavimo sprendimais grindžiama išplėstinė programų sistemų architektūra

Dušan SAVIĆ, Dejan SIMIĆ, Siniša VLAJIĆ

Programų sistemos architektūros parinkimas yra viena iš svarbiausių programinės įrangos projektavimo veiklų. Prieš pradedant projektuoti konkrečią sistemos struktūrą ir jos elgseną, reikia nuspręsti kokia bus jos architektūra. Straipsnyje pasiūlyta trijų lygmenų architektūra. Šioje architektūroje dalykinės srities logiką aprašantis lygmuo yra išplėstas apsaugos mechanizmais. Siūlomoje architektūroje realizuoti du svarbūs apsaugos procesai – autentifikavimas ir autorizavimas. Šie procesai realizuoti panaudojant tipinius projektavimo sprendimus. Siūloma programų sistemų architektūra apskritai yra grindžiama tipinių projektavimo sprendimų panaudojimu. Straipsnyje aprašyti siūlomi tipiniai apsaugos projektavimo sprendimai ir parodyta šių sprendimų panaudojimo svarba aprašomajai architektūrai.