Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Extending UWE with Secure Navigation Paths

Roman Schwienbacher

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Marianne Busch
Abgabetermin: 18. September 2012

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 18. September 2012

.....

Roman Schwienbacher

Abstract

The guarantee and implementation of data protection in modern web applications has challenged software engineers for several years. The first question that arises is which security aspects must be considered. In a second step, one must find a way to model them in a reasonable way in the design phase and, subsequently, apply them in the implementation phase of the software development process.

The issue of authentication and access control is one of the most important ones in the field of data protection. A new approach to ensure navigational access control is outlined by Secure Navigation Paths (SNPs). Within such navigation paths, a web application user with a certain role is only permitted to follow a limited number of paths in the intended order. This should protect both the user from unintended incorrect application procedures as well as the system from unauthorized attacks.

The core of this thesis is represented by the development of the, to the extent of our knowledge, first possibility to model SNPs. We develop this modeling approach by using the *Navigation State Model* of the UML-based Web Engineering (UWE) approach. UWE has been developed at the Institute of Programming and Software Engineering of the Ludwig-Maximilians-Universität Munich. It is a powerful method in the modeling of complete web information systems. Additional comfort is provided by our new Computer-Aided Software Engineering (CASE) tool plugin *MagicSNP*. Basically, this plugin allows to validate the designed security model and to extract the corresponding navigation rules. Therefore, it facilitates the handover between the modeling and the implementation progress of the application development. The last innovation presented in this thesis is our generic monitor module. This module is capable to provide Role-Based Access Control (RBAC) considering SNPs for JSF-based web applications.

In addition, the applicability and, furthermore, the reliability of our overall approach is demonstrated by a case study called *TicketApplication*. Originally, *TicketApplication* was a simple web application without access control management. Using our modeling approach, we design the RBAC behavior including the appropriate modeling of SNPs under consideration of the given use cases. Then we apply our monitor module which provides access control for our *TicketApplication* based on the navigation rules extracted by *MagicSNP*. As a result, we get a secure and robust web application, which fulfills the security standards of modern web applications. Therefore, the approaches of this thesis should be concerned within the context of data protection in modern web applications.

Zusammenfassung

Die Gewährleistung und Implementierung des Datenschutzes in modernen Webanwendungen stellt Softwarearchitekten seit mehreren Jahren vor große Herausforderungen. Zunächst stellt sich die Frage, welche Sicherheitsaspekte berücksichtigt werden müssen. In einem zweiten Schritt muss ein Weg gefunden werden, diese in der Entwurfsphase vernünftig zu modellieren, sowie anschließend in der Implementierungsphase des Softwareentwicklungsprozesses umzusetzen.

Der Aspekt der Authentifizierung und Zugriffskontrolle ist einer der wichtigsten im Bereich des Datenschutzes. Sichere Navigationspfade stellen einen neuen Ansatz zur Gewährleistung von navigationsbasierter Zugriffskontrolle dar. Innerhalb solcher sicheren Navigationspfade ist es einem Benutzer mit einer gewissen Rolle nur erlaubt eine begrenzte Anzahl von Pfaden innerhalb der Webanwendung in der vorgesehenen Reihenfolge zu verfolgen. Diese Methode soll sowohl den Benutzer vor versehentlich falschen Applikationsabläufen als auch das System vor Angriffen schützen.

Der Kern dieser Arbeit besteht in der Entwicklung des, laut unseren Kenntnissen, ersten Ansatzes zur Modellierung von sicheren Navigationspfaden. Wir entwickeln diesen neuen Modellierungsansatz unter Verwendung des *Navigation State Model* vom UML-based Web Engineering (UWE) Ansatz. UWE wurde am Lehrstuhl für Programmierung und Softwaretechnik der Ludwig-Maximilians-Universität München entwickelt und ist ein mächtiges Werkzeug für Softwarearchitekten im Bereich der Modellierung von kompletten Webanwendungen. Zusätzlicher Komfort wird von unserem neuen Computer-Aided Software Engineering (CASE) Tool Plugin *MagicSNP* geboten. Dieses ermöglicht die Validierung der erstellten Sicherheitsmodelle und die Extraktion der entsprechenden Navigationsregeln. Dadurch wird der Übergang zwischen der Modellierung und der Implementierung der Anwendung erleichtert. Die letzte Innovation der vorliegenden Arbeit stellt ein generisches Monitor Modul dar. Dieses ist fähig, Rollen-basierte Zugriffskontrolle (RBAC) unter Berücksichtigung sicherer Navigationspfade für JSF-basierte Webanwendungen zu gewährleisten.

Zusätzlich wird die Anwendbarkeit und vielmehr die Funktionsfähigkeit unseres Ansatzes anhand einer Fallstudie namens *TicketApplication* konkret demonstriert. *TicketApplication* war ursprünglich eine einfache Webanwendung, welche keine Zugriffskontrolle vorsah. Basierend auf den gegebenen Anwendungsfällen und unter Verwendung unseres Modellierungsansatzes modellieren wir das RBAC Verhalten inklusive sicherer Navigationspfade. Danach wird unser Monitormodul für die Webanwendung konfiguriert, wel-

ches basierend auf den von *MagicSNP* extrahierten Navigationsregeln Zugriffskontrolle für die Anwendung gewährleistet. Als Ergebnis erhalten wir eine sichere und robuste Webanwendung, welche die Sicherheitsstandards von modernen Webanwendungen erfüllt. Somit sollten die Herangehensweisen dieser Arbeit im Kontext von Datenschutz im modernen Webanwendungen mit berücksichtigt werden.

Acknowledgements

I would like to thank Prof. Dr. Martin Wirsing for giving me the opportunity to write this bachelor thesis. I would especially like to thank the supervisor of my thesis, Marianne Busch. Her professional and constructive feedback helped me to improve the quality of my thesis. Furthermore, I want to thank Martin Ochoa from *Siemens AG Corporate Research and Technologies* for the kind support. Finally, I would like to thank my reviewers and friends Birgit Haller and Klaus Rzehaczek.

Contents

Chapter 1

Introduction

In the era of modern information technologies, where sensitive personal data is stored and managed over the Internet on databases or remote servers, software engineers are faced with new tasks in the field of security and user guidance.

Since broadband Internet is available for almost anyone, anytime and anywhere, the demand and the availability of web applications has dramatically increased. The success of web applications is fundamentally based, among other factors like high-fidelity and user-friendliness, upon the guarantee of data protection. It creates confidence to the web user and enables protected data transfers of sensitive data over a basically unsafe medium like the Internet. Therefore, new security frameworks have been designed and developed to better protect web applications from unauthorized access. In general, security frameworks already apply the four pillars of security and data protection for web applications:

i. *Authentication* (the process of proving the identity of an user to gain access to a protected resource)

ii. *Authorization / Access control* (the process of authorization determines what a subject (e.g., an user or a program) is allowed to access, especially what it can do with specific objects (e.g., files) [And08])

iii. *Encryption* (the conversion of data into a form that cannot be understood by unauthorized subjects)

iv. *Session management* (the process of keeping user-specific application data during he is authenticated to the system)

Examples of such security frameworks include Spring Security[1] and Apache Shiro[2]. Both were designed to provide a high standard of security as comfortable as possible for already implemented web applications. Therefore, they gained popularity and they are used and recommended by a large community of application providers.

However, there is one important aspect of security in the area of data protection which has not been considered concretely yet: A kind of navigational access control which guarantees that every web user with a certain role has only a limited number of navigation paths inside the application context. We call them Secure Navigation Paths (SNPs). Suppose a web application procedure managed to open an online-banking-account, which consists of about ten steps. What happens when a user,

[1] Spring Security. `http://static.springsource.org/spring-security/site/`, last visited 2012-07-08
[2] Apache Shiro. `http://shiro.apache.org/`, last visited 2012-07-07

whether intentionally or not, jumps from step two, "indicating the personal data", to the last step, "confirmation", simply by calling the corresponding URL, and confirms the transaction? This can not only lead to fatal inconsistencies on the application state, but may also cause a worst case scenario like losing money or damaging the image of the application provider.

Regarding the addressed issue, the goal of this thesis is to fill the gap of missing possibilities to model and control SNPs to enhance data protection within web applications. We start by developing a first approach on how to design SNPs. In addition, we implement an innovative and generic monitor module which is capable to provide Role-Based Access Control (RBAC) considering SNPs for JSF-based web applications. Thus safety-critical jumps through the different context views of the application, called navigation nodes, should be avoided. Finally, we provide a new plugin for the Computer-Aided Software Engineering (CASE) tool MagicDraw[3]. Basically, this plugin validates the designed security models which contain SNP semantics and it is able to extract the corresponding navigation rules.

In order to develop our approach on how to design access control considering SNPs we decided to use the UML-based Web Engineering (UWE)[4] approach. UWE has been developed at the Institute of Programming and Software Engineering of the Ludwig-Maximilians-Universität Munich and is a powerful method in the modeling of complete web information systems. In general, UML is powerful enough to cover the requirements that arise when modeling web applications [KK11]. In addition, UWE extends the UML profile by a large set of useful security and web features. Basically, we use UWE's *Navigation State Model*, originally an UML state machine, to be able to design RBAC for web applications considering SNPs using states and transitions.

Our new Computer-Aided Software Engineering (CASE) tool plugin for MagicDraw is a tool to validate the designed security model and moreover to extract the corresponding access-control-semantics. By iterating recursively through all hierarchical states and by analyzing the incoming transitions, state names and tags, this tool fetches and converts the relevant information into a semi-structured data format like JSON. Consequently, this plugin secures, accelerates and reduces the complexity of the handover between the modeling and the implementation progress of the web application development.

In order to be capable to provide RBAC with SNPs for web applications considering the modeled access-control-semantics we develop a new generic monitor module approach. This module is responsible to decide whether or not an user is allowed to get access to a protected resource. The decision making is based on the web user's session information (e.g., previously visited location, assigned user roles etc.) and the extracted access-control rule file, generated by our new MagicDraw plugin. In order to ensure robustness, this monitor module also handles any kind of access-constraint violation: The web user gets redirected to a corresponding error-page including an appropriate error-message with possibility to go back to its previously visited navigation-context.

The whole approach is reliable, flexible, straightforward for software engineers as well as for programmers and increase the security and robustness for web applications. Finally, these quality factors are proven by a case-study where we develop a sample web application: By using our new modeling approach, we design RBAC considering SNPs in the design-phase. Then we use our new MagicDraw plugin to validate our security-model and to extract the corresponding navigation-rule file. Finally, we apply

[3]MagicDraw. `http://www.magicdraw.com/`, last visited 2012-07-10
[4]UWE. `http://uwe.pst.ifi.lmu.de`, last visited 2012-07-29

our generic monitor module and inject the navigation-rule file in the implementation-phase. As a result we get a secure and robust web application which excels the security standard of modern web applications by using and applying our approaches developed in this thesis.

The remainder is structured as follows: In chapter 2 we analyze the concepts of UML-based security modeling approaches and we introduce the fundamentals of how existing and established security frameworks provide security for web applications. Chapter 3 presents our new modeling approach inclusive our new CASE tool plug-in for MagicDraw. In chapter 4 we give an insight into the implementation of our generic monitor module. Chapter 5, shows our case study *TicketApplication*. Finally, Chapter 6 summarizes the results and provides an outlook on future work.

Chapter 2

Related Work and Technical Possibilities

For a solid understanding of the design part and the security-implementation shown in this thesis, it is important to take a deeper look on some already existing and well proved security modeling approaches and frameworks. First, we provide an outlook on what informational security aspects already can be modeled. Second, we figure out what kind of data is needed for the configuration of security frameworks and how these frameworks finally provide security for web applications. Our new modeling and monitoring approaches introduced in this thesis were build based on these fundamentals.

2.1 Security Modeling

The Unified Modeling Language (UML) is well-known in software engineering fields and proven as the de-facto standard for designing software systems. Therefore, our security modeling approach relies on an UML-based environment. The following section provides a short overview of already existing UML-based security modeling approaches.

2.1.1 UWE

UWE has been developed at the Institute of Programming and Software Engineering of the Ludwig-Maximilians-Universität Munich to provide a new UML based approach on how to design complete web applications. Basically, it defines the following five views for the separation of concerns:

i. *Content*: Specifies the used data structure of the web application, represented as UML class diagrams.

ii. *Role model / Basic Rights Model*: UWE's *Role Model* allows one to define characteristics of the user-groups with the purpose of authorization and access control. Based on this, the *Basic Rights Model* offers a compact notation for domain specific RBAC declarations. It shows all available user roles in combination with the managed data domains of the future web application. The model itself is an UML class-diagram whereas the domains are represented as class definitions and the user roles are specified by class instances of the class *Role*. The primary goal of this model is to declare which user role is allowed to access which attribute or perform which action inside a domain. Therefor, the user roles are connected to the corresponding domains/methods/attributes with stereotyped dependencies.

iii. *Navigation*: Specifies the navigational view of the web application including role-based navigational access control (e.g., the parts of webpages that are accessible by a certain user role). This model is designed as UML state machines and called *Navigation State Model*. The navigational view consists of possible hierarchical states, the navigation nodes of the web application. More detailed information about navigation state models is given in chapter 3.

iv. *Presentation*: Specifies the application views and their nested forms and widgets, including content validation checks and behavior specifications of interactive elements (e.g., auto-completion for input text fields). This concern is represented as a rough presentation model.

v. *Process*: Specifies the process structure as class diagrams. The workflow of the processes can be modeled with UML activity diagrams.

Considering these points, this method represents a powerful and dynamic modeling environment to design a complete web application ready to implement. Therefore, our security design approach is an extension of UWE which affects the *Navigation* part.

2.1.2 SecureUML

SecureUML [LBD02] provides an UML-based modeling language that allows one to model RBAC for actions on protected resources. Additional authorization constraints which depend on dynamic properties of the system state can be defined with the Object Constraint Language (OCL). Thereby, it is possible to define dynamic security constraints like "method *getCustomers()* is only available to role *employee* during weekdays".

Essentially, SecureUML is a language for specifying access control policies by modeling hierarchical roles, permissions, actions, resources and authorization constraints. However, in contrast to UWE's *Basic Rights Model*, SecureUML cannot express exceptions and requests to specify all permissions separately in association classes.

2.1.3 UMLSec

UMLSec [Jür04] provides an UML extension that enables the application developer to design model-driven:

i. *RBAC* on resources and actions

ii. *Guarded access* on system components like Java class-instances

iii. *Authenticity freshness*, *secrecy* and *integrity* for data streams

iv. *Secure information flows* to design possible information flows by state machines

However, compared to UWE models, UMLSec models are very detailed and therefore very complex. Another disadvantage is faced by the fact that UMLSec only supports UML 1.4.

2.1.4 ActionGUI

ActionGUI [BCE11] is a language for modeling Graphical User Interfaces (GUIs) for data-centric applications with access control policies. It allows one to design *widgets*, which are the basic elements of a GUI (e.g., textfields, buttons, comboboxes, checkboxes etc.) in combination with their possible *events* (e.g., clicking on a widget, entering a widget) which may trigger a set of associated *actions*. It is possible to define specific conditions for every available event or action. These conditions may depend on the information kept in the widgets itself or stored in the database.

In particular, an ActionGUI model consists of three models:

i. *Data model*, specifies the data structure of the application

ii. *Security model*, specifies the access control policies by roles, permissions and constraints

iii. *GUI model*, specifies the GUI of the application

2.2 Security Frameworks

The following selection contains a set of web security frameworks which allow to realize and implement the security aspects modeled by the presented notations of the previous section. It should be noted that this is currently not yet an automatic task. The developers still have to interpret and implement the designed semantics accurately and manually which is a time-consuming and error-prone task.

2.2.1 Apache Shiro Web-Features

Apache Shiro[1] is a security-framework inter alia for Java-web applications that complements them in the following points:

i. *Authentication* (logins are supported across one or more dynamic data sources like Java Database Connectivity (JDBC) or Lightweight Directory Access Protocol (LDAP))

ii. *Authorization / Access control* (based on roles or fine-grained permissions)

iii. *Encryption* (provided by a large set of hashing and cipher features)

iv. *Session management* (managed within an individual session storage like an enterprise cache, a relational database or proprietary data store)

In a nutshell, the developer only has to configure so called *Realms* persistently, whereby Shiro can decide which user has with which role on which sites with which rights access or not. Therefore, already existing web applications can be secured with Apache Shiro without much effort.

[1] Apache Shiro Web-Features. `http://shiro.apache.org/web-features.html`, last visited 2012-07-07

2.2.2 Spring Security

Spring Security[2] (former ACEGI) is a security-framework for Java-web applications, which provides authentication and access control. Users including their roles and access patterns can be declared by static or dynamic database driven user-services. They provide the basis for decision making of which user role has access to which sites or elements. If there is a user without authentication requesting a resource, he will be redirected to the login-context automatically. After a successful login and if the user is authorized to access the requested page, he will be redirected to the initially requested page.

The whole framework can be included and configured for already existing web applications without much effort.

2.2.3 jGuard Web

jGuard[3] is a Java library especially designed to equip Java-web applications with authentication and authorization. The following domains are available to configure access control features:

i. *jGuardFilter* (login, violation and logout page)

ii. *jGuardAuthentication* and *jGuardAuthorization* (scope of the authentication, encryption)

iii. *jGuardUserPrincipals* (user and role information)

iv. *jGuardPrincipalsPermissions* (which user and role has access on which page)

2.3 Summary

In order to model security, especially navigational access control, information like the user role in combination with the corresponding access-information and navigational-behavior like "*what happens if not authenticated?*" or "*what happens if not authorized?*" are of crucial importance.

It should be noted that adding such features to already implemented web applications is an error-prone task. Security aspects are part of the application behavior and should be considered right from the beginning of the application development process. Otherwise, undesirable behavior or serious security leaks may occur as a result of missing meticulous testing of the security behavior in combination with the residual application behavior.

Therefore, the modeling approach in this thesis enables web engineers to model security issues in an earlier phase of the application development process as the implementation phase (e.g., the design phase).

[2]Spring Security. `http://static.springsource.org/spring-security/site/`, last visited 2012-07-08
[3]jGuard Web. `http://jguard.xwiki.com/xwiki/bin/view/Main/WebHome`, last visited 2012-07-08

Chapter 3

Modeling Secure Navigation Paths with UWE

In this chapter, we take a look on how navigational access control already can be modeled in UWE using its *Navigation State Model*. Afterwards, we introduce our new modeling approach that allows one to model access control by taking Secure Navigation Paths (SNPs) into account. Finally, we present our new plugin *MagicSNP* for the Computer-Aided Software Engineering (CASE) tool MagicDraw[1]: By performing one simple mouse click, this plugin is able to validate the selected model and moreover to generate a corresponding rule-file containing the data records, required to provide access control with SNPs.

3.1 Navigation State Model

A navigation state model describes the navigation structure of a Web-Information-System (WIS) and its behavior according to the different states. In UWE, navigation can be represented by a UML state machine: States, possibly hierarchical, represent navigational nodes, transitions the navigational links between the nodes. [BKK11]

Every state can hold a tag named *roles* that defines which role instances are allowed to enter the corresponding state. Another optional tag called *unauthorizedAccess* specifies which state a user will be redirected to if his role is not authorized to navigate to the requested navigational node. The last important boolean tag of a state is named *isHome*. It defines the login-context where the user will be redirected to in case he is not authenticated.

In order to design the *Navigation State Models* of this thesis we use the existing MagicDraw plugin *MagicUWE* [BK09]. *MagicUWE* is a CASE tool for designing and generating UWE models characterized by its usability.

3.2 Our Modeling Approach

UWE's *Navigation State Model* already covers all access control based constraints for web applications which can be monitored by the presently existing security frameworks shown in chapter 2. In order to design SNPs we simply use the transitions between the states. They already provide all the required means for specifying SNPs. Applying this to the *Navigation State Model* illustrated in Figure 3.1 it cannot be possible that

[1]MagicDraw. `http://www.magicdraw.com/`, last visited 2012-07-10

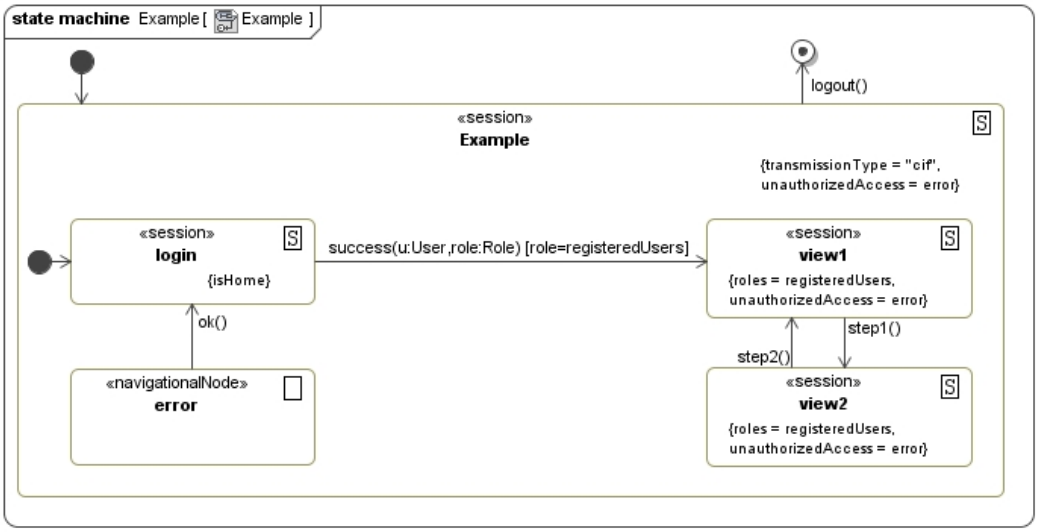

Figure 3.1: *Simple Navigation State Model*

a user stays on the navigation state *view2* without having been on the navigation state *view1* right before. This constraint is indicated by the outgoing transition pointing from *view1* to *view2*. Therefore, the redirect to some violation state (defined in an *unouthorizedAccess* tag) occurs not only if the role is not authorized to jump to it corresponding to the RBAC. It also occurs in case the user was not on a state that has no outgoing transition to the requested state right before.

3.2.1 Constraints

Every *Navigation State Model* has to contain one parent state which represents the whole web application. On this level the tag *unauthorizedAccess* is mandatory. It passes the default violation information to its substates in case they have no such tag defined.

In order to provide the information which navigation node represents the login-context, the whole model must exhibit exactly one navigation state declared as home state by the tag *isHome*.

3.2.2 Recommendations

Nesting increases the readability of the *Navigation State Model* and avoids multiple tag declarations and transitions. Therefore, nesting is a recommended pattern to structure and group navigational nodes to. Theoretically, it is allowed to nest the navigation states until any level. However, the grade of this level should not be higher than five or six to keep the readability on an acceptable level. All the tags, declared in a parent state, are passed to the corresponding child states, but can also be overwritten by redefining them on a lower level. It should be noted that the *isHome* tag will be passed to the innermost initial state only as well as this state will inherit all the incoming transitions of its ancestor states. Therefore, it can be said that only states without child states are representing real navigation nodes inside the future application. In order to differentiate them from parent states, we decided to use the convention of labeling their name with the first letter in lower case.

Generally, it is strongly recommended that violation navigation states, declared in any *unauthorizedAccess* tag, should not exhibit incoming transitions to avoid non terminating violation redirects.

The tag transmissionType="cif" is optional, it sets the overall type of data transmission during the session to cif, providing for confidentiality, integrity, and freshness: The implementation should prevent eavesdropping, replaying, or altering of transmitted data. [BKK11]

The stereotype *session* just declares that several session information is kept on the browsing user entering this navigation node.

Last but not least, the call event *logout()* leads to the termination of the *Navigation State Model*. This event does not stand for the end of the whole application. Moreover, it represents the end of one single user-session context where every user-specific information corresponding to the web application will be removed. In other words *logout()* just leads to the logout of the corresponding user.

3.3 Representation of Secure Navigation Paths in Plain UML

The precise meaning of UWE's *Navigation State Model* including the described extensions for SNPs can be illustrated by a transformation into plain UML.

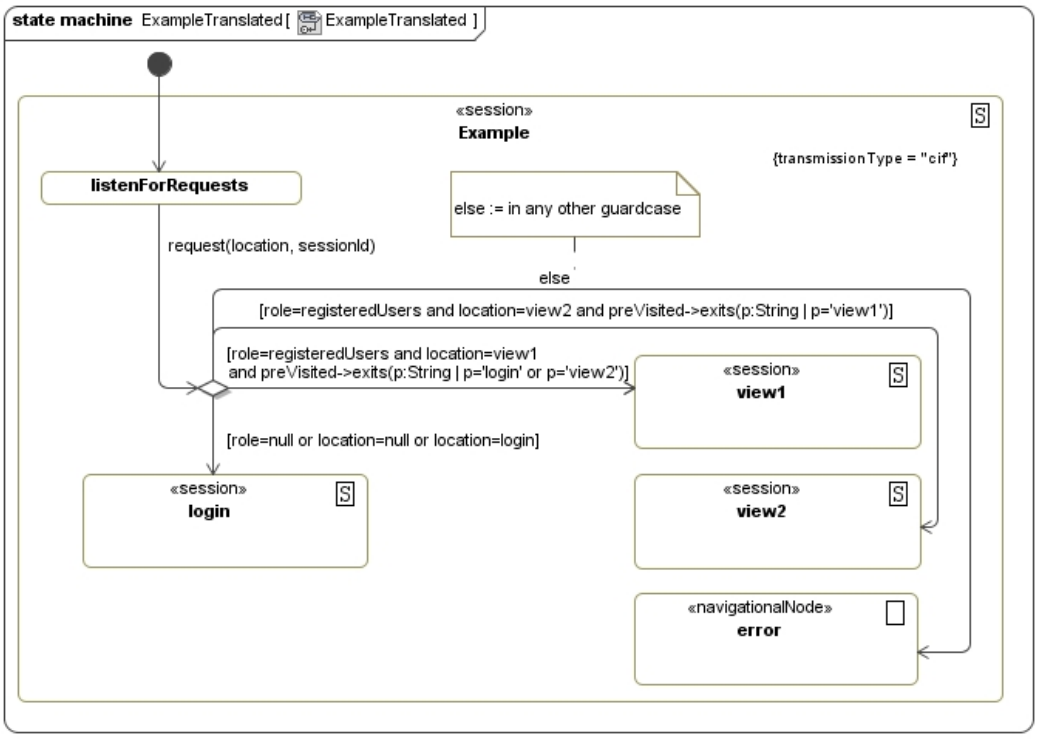

Figure 3.2: *A transformation into Plain UML*

As depicted in Figure 3.2, there is one new state *listenForRequests* which handles every request on the web application. Every real navigation state is reachable from this state, controlled by guard conditions. These are generated by translating the session tags *roles*, *unauthorizedAccess* and the implied SNP information out of the transitions.

In case the user is logged in, his role and last visited navigation node are stored in the session attributes *role* and *preVisited* identified by the sessionId of the request. The requested navigation node comes as parameter *location* through the request context. The guards are checking exactly these attributes. However, the default unauthorized access state gets the *else* guard-condition.

3.4 Supporting CASE Tool Plugin *MagicSNP*

Once the *Navigation State Model* is designed and valid according to the requirements mentioned in the section 3.2.1, it takes great effort to apply the whole system of rules to an independent security provider of a web application. Rather than implementing rule by rule it would be a great benefit to have the possibility to generate the whole system of rules directly right out of the *Navigation State Model*. However, that is exactly what the MagicDraw plugin *MagicSNP* does. The main goal of this plugin is to provide a convenient tool for securing, accelerating and reducing the complexity of the handover between the modeling and the implementation progress of the whole application development. In addition, the developer gets a robust tool to validate *Navigation State Models* regarding to the modeling constraints and recommendations defined in this chapter.

3.4.1 Usage of *MagicSNP*

Figure 3.3 depicts how to call the plugin *MagicSNP*:

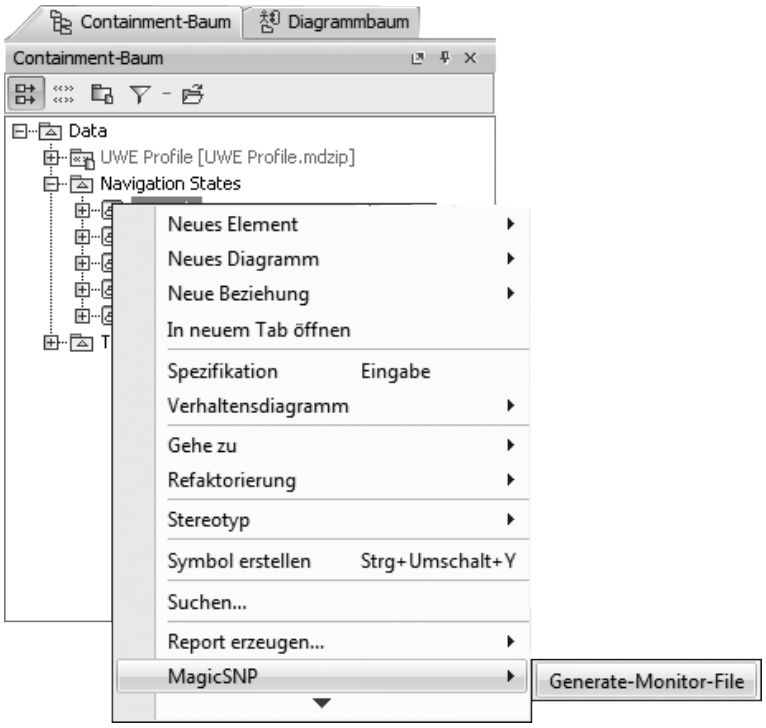

Figure 3.3: *Using the MagicDraw plugin MagicSNP*

Just perform a right click on the *Navigation State Model* in the containment-tree to open the context menu and select *MagicSNP/Generate-Monitor-File*. Right after that a pop-up window opens which contains all the semantic information of the selected

Navigation State Model context. In case the designed model is not valid, the pop-up simply contains the corresponding error message. Therefore, this plugin can also be used or seen as a quick validation tool.

The content (except the error message) is semi-structured as JavaScript Object Notation (JSON)[2] and conforms to the Java properties[3] format. Assuming the security backend is implemented in Java and reads the security configuration parameters out of a Java properties file: This format ensures a quick transfer of the generated data between the modeling and the implementation context by performing a simple copy-paste.

3.4.2 Implementation of *MagicSNP*

We developed *MagicSNP* using MagicDraw's open API[4]. In order to get a better understanding of how the plugin collects and finally builds the data from the selected model, this subsection will describe its entire process sequence. Figure 3.4 illustrates the activity sequence of the rule extracting job on an abstract level. It starts by checking if the state machine is valid corresponding to the following two constraints:

i. There must be exactly one parent application state representing the whole application which exhibits the default *unauthorizedAccess* definition

ii. There must be exactly one navigation state which is declared as home state

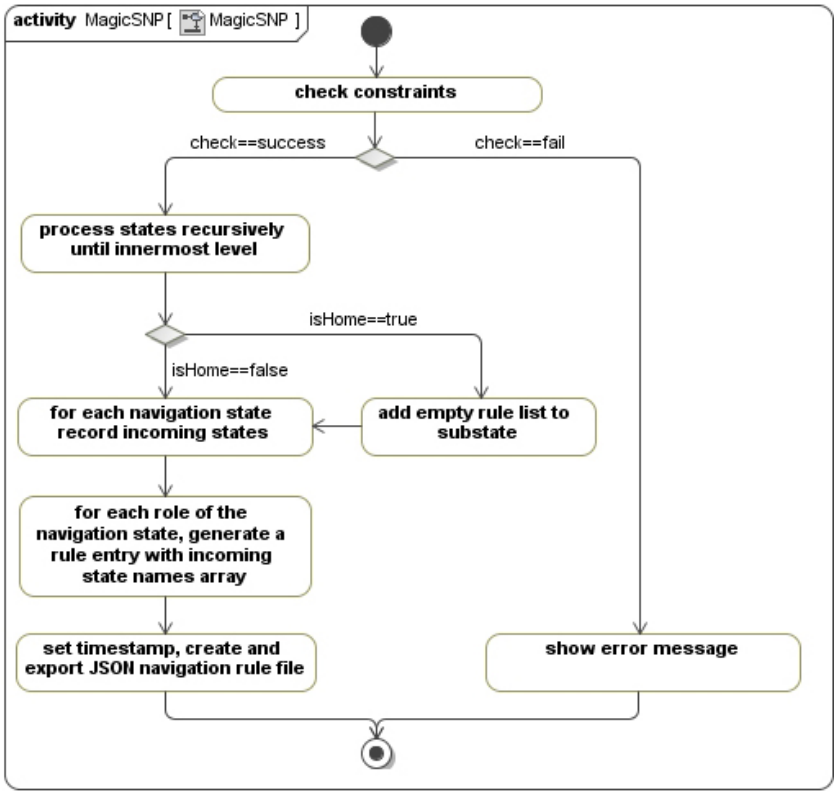

Figure 3.4: *Custom MagicDraw plugin MagicSNP: activity diagram*

[2]JSON. `http://www.json.org/`, last visited 2012-07-03
[3]Java properties. `http://docs.oracle.com/javase/6/util/Properties.html`, last visited 2012-07-03
[4]Open API. `http://www.nomagic.com/files/manuals/MagicDraw%20OpenAPI%20UserGuide.pdf`, last visited 2012-08-02

If one of these constraints fails, the job terminates immediately and returns just an appropriate error message.

Otherwise, the job runs in hierarchical order recursively through all states until the innermost level, the level of the navigation states, is achieved: For each one of these, the *roles* and the *unauthorizedAccess* information is collected considering the inheritance rules of UML state machines. Finally, every navigation state of the future application is represented by one *location* entry. All the incoming navigation state names are recorded and written as *pre_visited* into the *rules* attribute of the current *location* entry to represent one possible step of an SNP. In case of an incoming state is a parent state, the job considers all the child states on the innermost level instead of the state itself. If a navigation state is a child state and at the same time the initial state inside his level, all assigned incoming state names of his parent were also assigned.

For each role defined for a navigation state, the *rule* attribute inside a *location* record will be duplicated containing the corresponding role information, with the exception of the *isHome* state. The login context should be available for every surfer regardless of its session context. Therefore, an empty rule list is added to the entry for this state.

Finally, a time stamp is set to get a history inside the navigation rule files. Afterwards, the job puts all the information together and generates a JSON structured navigation rule file. This can be taken as input for a security framework which provides RBAC including SNPs for the modeled web application. Figure 3.5 depicts such a rule file based on the *Navigation State Model* illustrated by Figure 3.1.

```
1  navigation.file={ \
2    "_comment":"Build time: 05.06.2012 09:28:56",\
3    "application":"Example",\
4    "locations":[\
5      {"location":"login", "violation":"error", "home":true,\
6       "rules":[{"role":"*","pre_visited":[]}]},\
7      {"location":"view1", "violation":"error", "home":false,\
8       "rules":[{"role":"registeredUsers","pre_visited":["login","view2"]}]},\
9      {"location":"view2", "violation":"error", "home":false,\
10      "rules":[{"role":"registeredUsers","pre_visited":["view1"]}]},\
11     {"location":"error", "violation":"error", "home":false,\
12      "rules":[{"role":"*","pre_visited":[]}]}],\
13   "default_violation":"error"}
```

Figure 3.5: *Extracted JSON structured navigation rule file*

The content corresponds to the Java properties format. There is only one property named *navigation.file* which holds the JSON structured navigation-rule data. The first attribute on the outermost level *_comment* holds the build time meta-information. The attribute *application* contains the application name and *default_violation* contains the default violation navigation node. The property *locations* contains an array of all possible navigation nodes of the future application. Every array item consists of a set of attributes:

i. *location* (stores the name of the navigation node)

ii. *violation* (specifies which navigation node is entered when an access rule is violated)

iii. *home* (identifies the login context node)

iv. *rules* (holds an array for every role which is permitted to access the current node)

Each item of the *rules* array consists of an attribute named *role* which stands for the user role and an array which holds all possible preceding navigation-nodes named *pre_visited*. The user with the corresponding role must have visited exactly one of these nodes right before to have a right to access the current *location*.

The value of the *role* attribute can hold the Kleene star which means that this rule applies for every request without considering the user role. In case the *pre_visited* array is empty, it does not matter on which navigation node the user with the corresponding role has been right before. Usually, the *location* entry which is tagged as *home* location has just one single *rules* item ({`"role":"*","pre_visited":[]`}) which grants that every user has access to it.

3.5 Summary

This chapter presented our new design pattern to model SNPs in UWE. In addition, our new MagicDraw plugin *MagicSNP* and its ability to generate a navigation rule file out of the designed *Navigation State Model* with one simple mouse click was introduced.

The following chapter shows our monitor module which is able to read and interpret this file dynamically. Basically, it represents the *listenForRequests* state, as depicted in figure 3.2, by monitoring the navigational access control for a web application corresponding to given navigation rules.

Chapter 4

Monitor Module

This chapter shows the conception and implementation of our new autonomous and generic monitoring module. It is capable to provide role-based navigational access control for JSF[1]-based web applications considering Secure Navigation Paths (SNPs). First, we present the class structure of the domains this module uses to calculate whether or not a specific request is allowed. Second, we introduce the components and sub-components of which this module consists. Additionally, we analyze which responsibilities they have and how they collaborate with each other. Third, we provide an instruction on how to prepare a web application to be monitored by this approach. Finally, we list the third-party frameworks we used to implement this module.

4.1 Rule Domains

The cleanest way to work with structured data inside a Java application is given by using object-oriented domain classes. Therefore, the module needs a specific object-oriented data structure which holds the information given by the JSON structured navigation rule file, generated in the design phase of the development process.

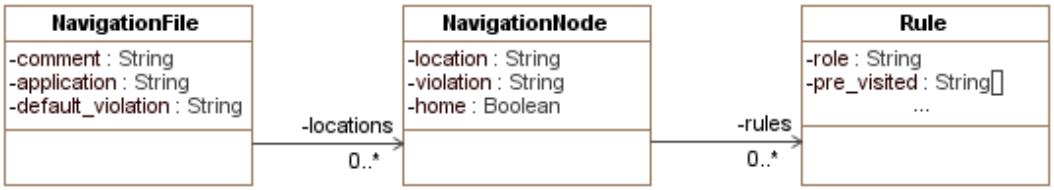

Figure 4.1: *Rule Domains: Class diagram*

Figure 4.1 depicts the structural dependencies of the rule domains `NavigationFile`, `NavigationNode` and `Rule`: One `NavigationFile` instance can hold a set of as many `NavigationNode` instances as needed (stored in the private locations attribute). Every object of the class `NavigationNode` can hold any number of `Rule` instances (stored in the private rules attribute).

[1]JSF. `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`, last visited 2012-07-18

4.1.1 `NavigationFile`

An instance of the class `NavigationFile` represents the whole navigation rule file. It contains the application name, build information, default violation and a set of locations represented by objects of the class `NavigationNode`.

4.1.2 `NavigationNode`

Every instance of the class `NavigationNode` matches to a view element of the web application. This class consists of the node name, the corresponding violation, the boolean attribute, whether it is the login node or not, and a set of `Rule` domains.

4.1.3 `Rule`

Every `Rule` object holds security based access information for one specific user role. It defines which possible navigation nodes could have been visited right before to be authorized to access the corresponding navigation node considering defined SNPs.

4.2 Module Components

As autonomous and generic access control framework, our monitor module acts as independent interface between web user and the web application. The first skill is to monitor every single request and to manage the communication between the corresponding web user and the web application. The second skill is to vote whether or not a user is allowed to access a requested protected location. In order to be capable to handle such request-voting simultaneously, it is necessary to have a kind of flow-control inside the framework. Therefore, our monitor module consists of two major components: Firstly, the *Security Provider* which processes whether or not a request on a specific resource is granted. Secondly, the *App Monitor* which handles every single request on the web application and schedules the voting requests to the *Security Provider* to avoid request overflows.

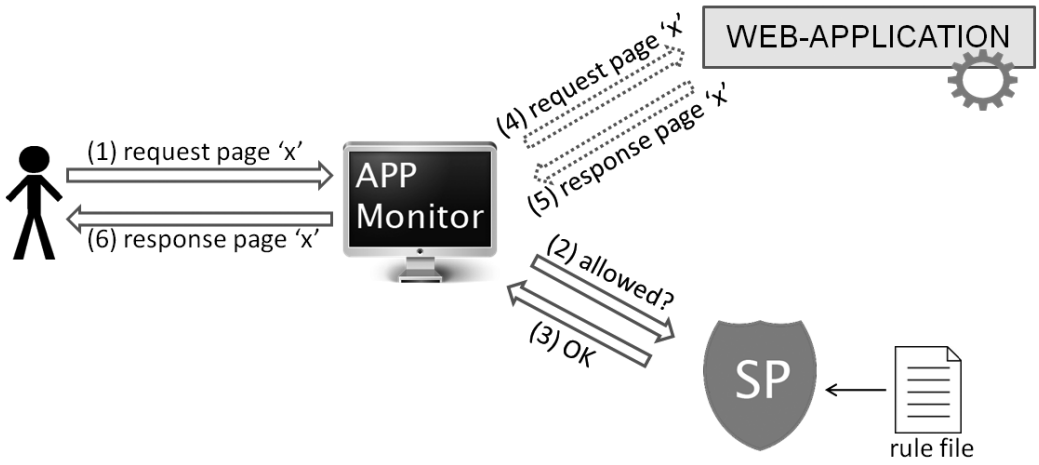

Figure 4.2: *Module Components: Request life cycle*

Figure 4.2 depicts the primary scenario of one life cycle for one user request which has to pass the access control of our monitor module: The web user requests the

protected resource *x*. The *App Monitor* reads the request parameters and asks the *Security Provider* whether or not the access is allowed. In this case the *Security Provider* allows the access corresponding to the rule file. Therefore, the *App Monitor* passes user-request to the web application and the application-response to the web user.

Further information about these major components is given in the following two subsections.

4.2.1 Security Provider

The class `SecurityProvider` is capable to decide through the `AccessVoter` whether or not a user is allowed to access a requested location with his pre-visited location and his roles. Additionally, this class can provide the login node or the corresponding violation node calculated by the `ViolationLocationProvider` in case of an access violation.

This service must be available during the whole runtime to ensure permanent navigational access control to avoid security leaks. Therefore, the class `SecurityProvider` loads the JSON structured navigation rule file content generated by the MagicDraw plugin introduced in chapter 3 just as the application server of the web application launches. Finally, the content is parsed to the static `NavigationFile` instance *navigationFile* of this class which corresponds to the object-oriented data structure of section 4.1.

```
1  // called @ spring initialization / server start-up
2  public void setApplicationContext(ApplicationContext ac) throws BeansException {
3
4    log.debug("try to parse " + S_FILE);
5
6    try {
7
8      // load properties resource
9      Properties properties = new Properties();
10     properties.load(SecurityProvider.class.getResourceAsStream("/" + S_FILE));
11
12     if (properties.containsKey("navigation.file")) {
13
14       // check if value is JSON valid
15       if (JSONUtils.mayBeJSON(properties.getProperty(NAV_PROPERTY))) {
16
17         // read JSON content out of property
18         JSONObject jsonHolder =
19           JSONObject.fromObject(properties.getProperty(NAV_PROPERTY));
20
21         // parse to NavigationFile instance
22         navigationFile =
23           (NavigationFile) JSONObject.toBean(jsonHolder, NavigationFile.class);
24
25         log.debug(S_FILE + " parsed successfully");
26         log.debug("Application name: " + navigationFile.getApplication());
27         log.debug("File generation date: " + navigationFile.get_comment());
28
29       } else {
30         log.error("Security file " + S_FILE + " contains no valid JSON");
31       }
32     } else {
33       log.error("Security property " + NAV_PROPERTY + " not found");
34     }
35   } catch (Exception _e) {
36     log.error("Error @parsing " + S_FILE, _e);
37   }
38 }
```

Listing 4.1: *Method setApplicationContext: called at server start-up, parses the navigation rule file*

The method `setApplicationContext`, depicted in Listing 4.1, is responsible for the load and parse topic we described above. This method consists of several logging commands and instructions to load the rule file. However, the major task is the act of parsing and initializing the central `NavigationFile` instance `navigationFile` at line 22 and 23. As result of the structural equality between the JSON structured rule content and our rule domains this task is covered by one single line of code using the static method `toBean` of the class `JSONObject`. This method is part of the *Maven JSON-lib*[2] API we use for our monitor module.

There is a disadvantage in loading the navigation rule file when the server launches: Every modification of the access control rules causes a relaunch of the application server. An alternative to this approach would have been to make the navigation rule file changable during the runtime without restarting the application server. That would increase the up-time of the web application. However, this approach would imply a negative impact on the performance of the web application, because the navigation rule file would be needed to be parsed before every access request. Therefore, we decided to do this job just once at the server startup.

The `AccessVoter` of this module is the logical interface between the design-phase with its generated rule file and our web application monitor module approach presented in this chapter. Basically, for the voting about some arbitrary user request it requires the following input parameters: Firstly, the assigned user roles. Secondly, the last visited navigation node inside the application context and finally the requested node.

When called, the component iterates through the `NavigationNode` instances of the `NavigationFile` object until the instance corresponding to the requested location is matched. Right after that the system searches the right `Rule` instance of the `NavigationNode` for the role information of the user (the Kleene star matches to all possible rules incidentally). If found, the given last visited navigation node of the user is syndicated by the possible navigation nodes regarding to the well-defined SNP. If at least one node can be matched, the returned voting result will be *true*. In the special case that the requested location is equal to the last pre-visited location, the voting result will also be *true* (e.g., a submit occurs on a certain page and an input value inside a form is not valid). In any other cases the voter votes *false* and the access to the requested resource would not be allowed.

In case the `AccessVoter` does not allow the access to a certain node, the system requires a violation node for redirecting to take the user into account, an access control violation occurred.

The `ViolationLocationProvider` of the `SecurityProvider` returns the violation entry of the given requested location configured in the corresponding `NavigationNode` instance. If the requested location cannot be matched to any instance or no violation node is defined for the corresponding navigation node, the application wide default violation, will be returned.

4.2.2 App Monitor

The class `AppMonitor` is the central singleton module which controls every navigational request to the application. It provides role-based navigational access control considering SNPs.

In order to work properly, this module deals with the following session-based attributes of every client:

[2]Maven JSON-lib. `http://sourceforge.net/projects/json-lib/files/`, last visited 2012-07-03

i. *username* (needed to generate user specific violation messages)

ii. *roles* (separated by ";" and needed for the `SecurityProvider.AccessVoter`)

iii. *pre_visited* (needed for the `SecurityProvider.AccessVoter`)

iv. *req_location* (needed to remember the requested location in the case the surfer is not logged in and is being redirected to the appropriate login context)

v. *message* (displayed message, shown on the violation nodes)

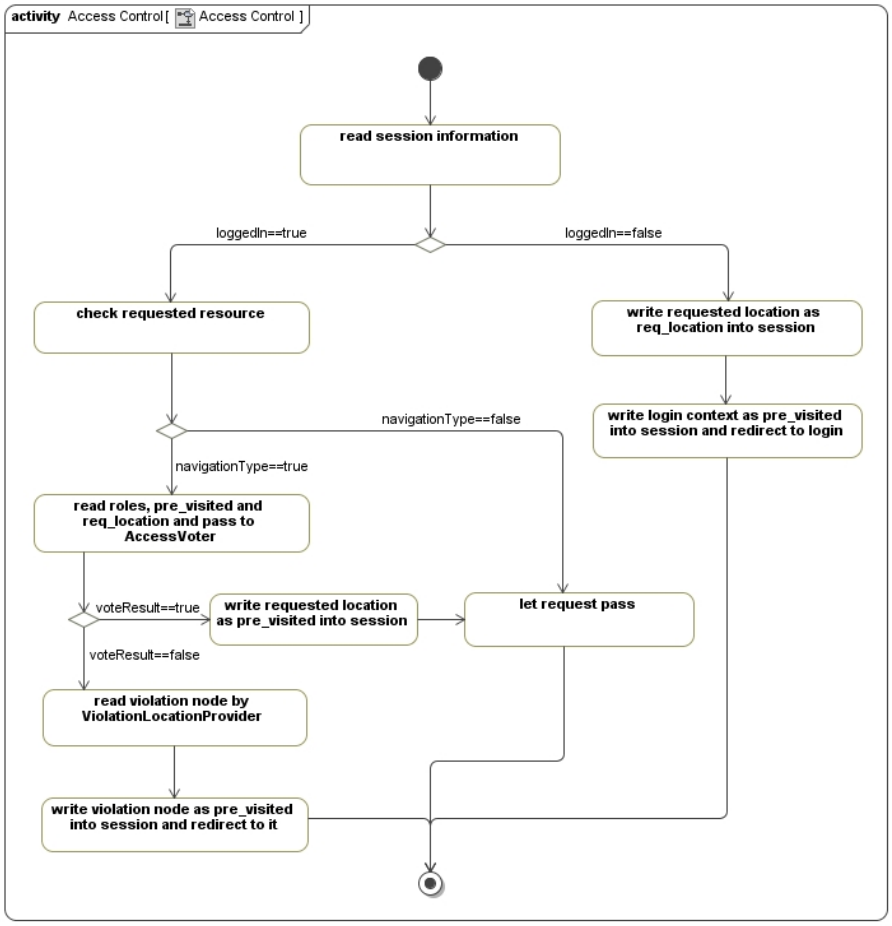

Figure 4.3: *`AppMonitor` activity diagram*

Figure 4.3 represents the execution of the `AppMonitor` component. The first check this component performs after a navigational request is whether or not the requesting client has at least one assigned role inside the application context. In other words, the login state is checked. If not logged in, the requested node will be written as *req_location* into the session and the user is redirected to the login context, provided by the `SecurityProvider`.

In order to be continuously aware of the last visited node inside the application navigation, this component performs a process which writes, just before a redirect occurs, the destination into the session as *pre_visited* attribute.

In case the user is already logged in, the monitor checks which kind of resource is requested. Requests without navigational contexts will pass (e.g., CSS-resources).

Every request with navigational context has to be checked whether it is allowed or not according to the `SecurityProvider`. Therefore, the process collects all the necessary data records like the user roles, the last visited node inside the application and of course the requested node. As described above, these parameters are passed into the `AccessVoter` of the `SecurityProvider` which votes for yes or no corresponding to the current request. If access is granted, the monitor redirects the user to the requested node immediately. Otherwise, the violation node is determined by the `ViolationLocationProvider`. Furthermore, an individual error message is generated and written as *message* attribute into the session so that the violation node, where the user is redirected to, can handle or present it to the user. It should be noted that this component reads the requested node value directly out of the called URL file definition (e.g., "http://www.myapp.com/appContext/protected/home.xhtml" leads to requestedNode=home). Of course, this behavior can be redefined without much effort e.g., to read this value corresponding to some request parameter inside the URL.

4.3 Preparing a Web Application to be Monitored

This section describes how an existing web application can be associated with the whole monitoring module presented in this chapter. In addition, we motivate which behavior and structure is recommended to guarantee a robust and user-friendly runtime.

4.3.1 Configuration

First of all, the classes of this module have to be available in the classpath of the web application-backend. The cleanest way to do this is to create and provide a Java Archive (JAR) of the monitor module.

The navigation rule file out of the design phase must be available on the outermost level in the classpath too, and has to be named *security.properties*. The usage of Java properties is proven as the de-facto standard for static and administrative Java system configurations. Therefore, we decided to use Java properties instead of an usual configuration file. Of course, the name and the exact position of this file can be modified inside the class `SecurityProvider`.

The simplest way to provide navigational access control for a web application would have been to monitor every single request and to decide whether the request is allowed or not. However, to safe performance it is much better to perform access control for navigation nodes which exhibit restrictions only (e.g., nodes with a role restriction or at least one incoming transition inside the *Navigation State Model* of the design phase). Therefore, they must have a directory named *protected* as last ancestor corresponding to the pattern `/protected/*/*.xhtml`.

```
1 <!-- SECURITY FILTER -->
2 <filter>
3   <filter-name>AppMonitor</filter-name>
4   <filter-class>de.package.name.AppMonitor</filter-class>
5 </filter>
6 <filter-mapping>
7   <filter-name>AppMonitor</filter-name>
8   <url-pattern>/protected/*</url-pattern>
9 </filter-mapping>
```

Figure 4.4: *Filter mapping snippet from web.xml deployment descriptor*

In addition, the AppMonitor must be registered on the URL-pattern `/protected/*` inside the deployment descriptor, mostly the *web.xml* as shown in Figure 4.4. It should be noted that the URL pattern can also be customized as required (e.g., by adding a special suffix to every protected node).

After these steps, the monitor module will be able to manage the navigational access control of the web application as described in this chapter. An alternative to this configuration would have been to deploy the monitor module as independent service. This approach would prevent the need to apply the configurations, introduced in this section, to the existing web application. All the configuration work would be done inside the monitor service. However, to work with common session attributes between two independent applications would force the usage of cookies. Cookies can be manipulated by the web user without much effort. Therefore, we decided to deploy the web application and the monitor module as homogeneous system to avoid cookie-based security leaks. Thus, our approach allows to store the session based information inside the application server.

4.3.2 Recommended Behavior

In case of a web user requests a protected resource without being logged in, he should be redirected to this resource after the login succeeds. Therefore, the managed JSF bean of the login context view should redirect the user to the node indicated by the session attribute *req_location* in case of a successful login.

Robustness and user-friendliness are two major factors for high-quality software. [BB78] Therefore, the managed bean of every violation node should read the session attribute *message* and make it visible through the frontend for the user.

4.4 Used technology

The whole module is written in Java EE 1.6.[3] The implementation of basic topics as well as Dependency Injection (DI), JSON parsing and logging does not influence the concept of this monitor module approach. Therefore, we decided to save a huge amount of implementation work by using the following third party libraries:

i. *Spring Framework*[4] (DI)

ii. *Maven JSON-lib*[5] (validation and parsing of JSON structured content)

iii. *Apache Commons*[6] (logging)

As alternative frameworks we could also have been using *Guice*[7] for DI, *GSON*[8] for JSON parsing and *SLF4J*[9] for logging.

[3]Java EE 1.6. `http://www.oracle.com/technetwork/java/javaee/downloads/index.html`, last visited 2012-07-08

[4]Spring Framework. `http://www.springsource.org/spring-framework/download`, last visited 2012-07-08

[5]Maven JSON-lib. `http://sourceforge.net/projects/json-lib/files/`, last visited 2012-07-03

[6]Apache Commons. `http://commons.apache.org/`, last visited 2012-07-10

[7]Google Guice. `http://code.google.com/p/google-guice/`, last visited 2012-07-24

[8]Google GSON. `http://code.google.com/p/google-gson/`, last visited 2012-07-24

[9]Simple Logging Facade For Java. `http://www.slf4j.org/download.html`, last visited 2012-07-24

4.5 Summary

The monitor module is intended to handle every navigational request inside a web application: In case the requesting web user is not logged this module ensures the redirect to the login context. In case the user satisfies the constraints written in the access-control rule file this module passes the request to the web application. In case of a constraint violation this module redirects the user to the corresponding error node.

As announced at the beginning of this chapter, this is just one possibility how to monitor a web application regarding the rules generated from the security modeling phase. The constitution was inspired by established frameworks like Spring Security and Apache Shiro. We discussed possible alternatives and corresponding advantages or disadvantages. In summary, our approach was built under consideration of security, robustness and performance aspects. Therefore, we were able to develop a sample web application introduced in chapter 5 which fulfills highest quality standards for software using our monitor approach.

Chapter 5

Case Study *TicketApplication*

As announced in the introduction part of this thesis, we introduce a new sample web application called *Ticket Application*. Thus we provide a complete example of how the techniques and approaches presented in this thesis can be applied concretely to a web application and how they depend on each other.

After introducing the primary use case of our case study, this chapter shows UWE's basic rights model corresponding to the web application. Furthermore, we present the modeling of the access control using UWE's *Navigation State Model* considering Secure Navigation Paths (SNPs). Right after that, we analyze the navigation rule file generated by the MagicDraw plugin *MagicSNP*. Additionally, we list the set of necessary points to apply our monitor module introduced in chapter 4 to the web application. Finally, we present the third party frameworks we used to implement the complete web application.

5.1 Use Cases

In order to get a better understanding of the application behavior, we should take a look to the corresponding use cases. They can be represented by the use case diagram depicted in Figure 5.1.

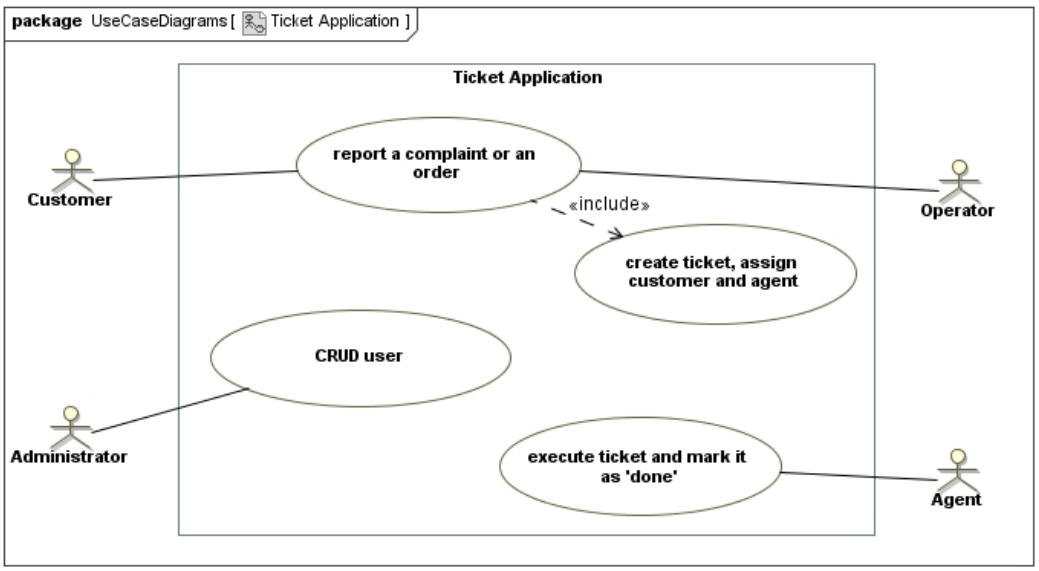

Figure 5.1: *TicketApplication: Use case diagram*

The ticket application should represent a kind of complaint and order management for any company and provides two roles namely *admins* and *registeredUsers*:

i. *admins* have the possibility to manage all stored user instances.

ii. *registeredUsers* have the permission to create or edit tickets.

Primary scenario (*registeredUsers*): Customers call the call center of the company. They describe their concern and thereupon the operator will create a new ticket. In order to do that, the operator must enter the customers identification number into the system in order to load the complete customer information from an external database. This customer instance will be assigned to the ticket. Right after that, the ticket nature gets specified. Finally, the operator has to confirm the modifications on a separate confirmation page.

Concurrently, the ticket is assigned to an agent, which opens it after the appropriate execution and marks it as *done*, followed by the confirmation.

5.2 Basic Rights Model

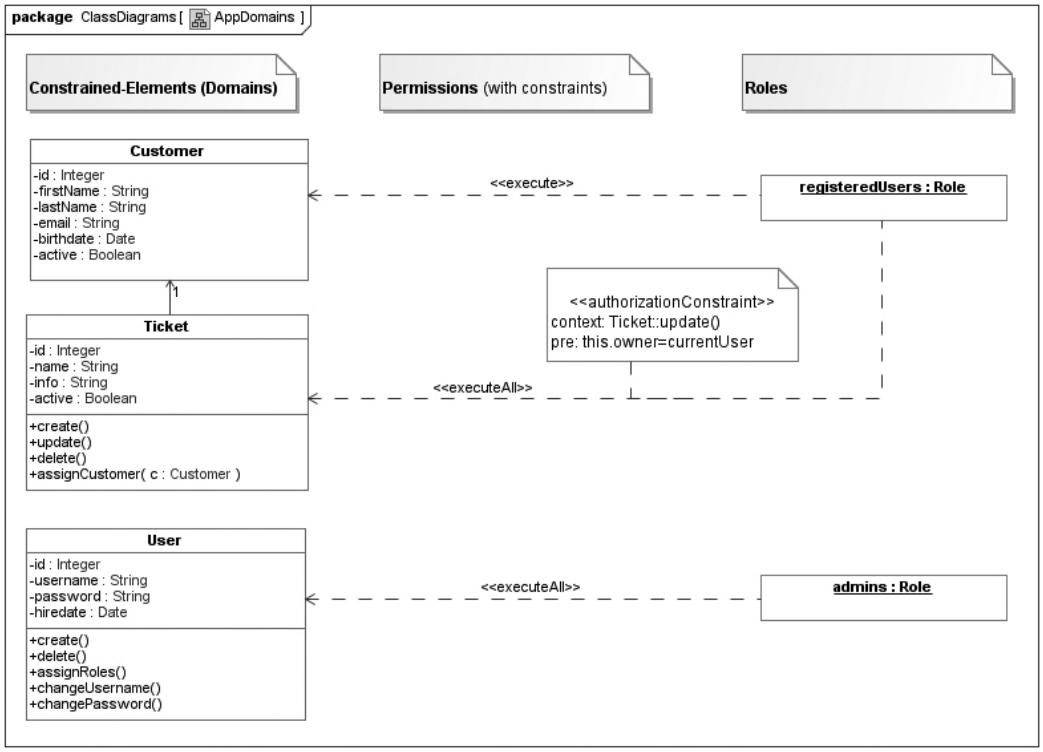

Figure 5.2: *TicketApplication: Basic rights model*

UWE's *Basic Rights Model* for the ticket application example, depicted in Figure 5.2, offers a compact notation for domain specific RBAC declarations. Basically, it shows all available user roles in combination with the managed data domains of the web application. Additionally, it specifies execution rights on methods with dependencies stereotyped «execute» and «executeAll». Further restrictions are defined in comments stereotyped by «authorizationConstraint» in the OCL: Users with the role

registeredUsers shall only be allowed to update a *Ticket* instance, when they are the assigned owner.

5.3 Navigation State Model

Taking the use cases of section 5.1 into account, we get the instance of UWE's *Navigation State Model* shown in Figure 5.3 considering SNPs as described in chapter 3.

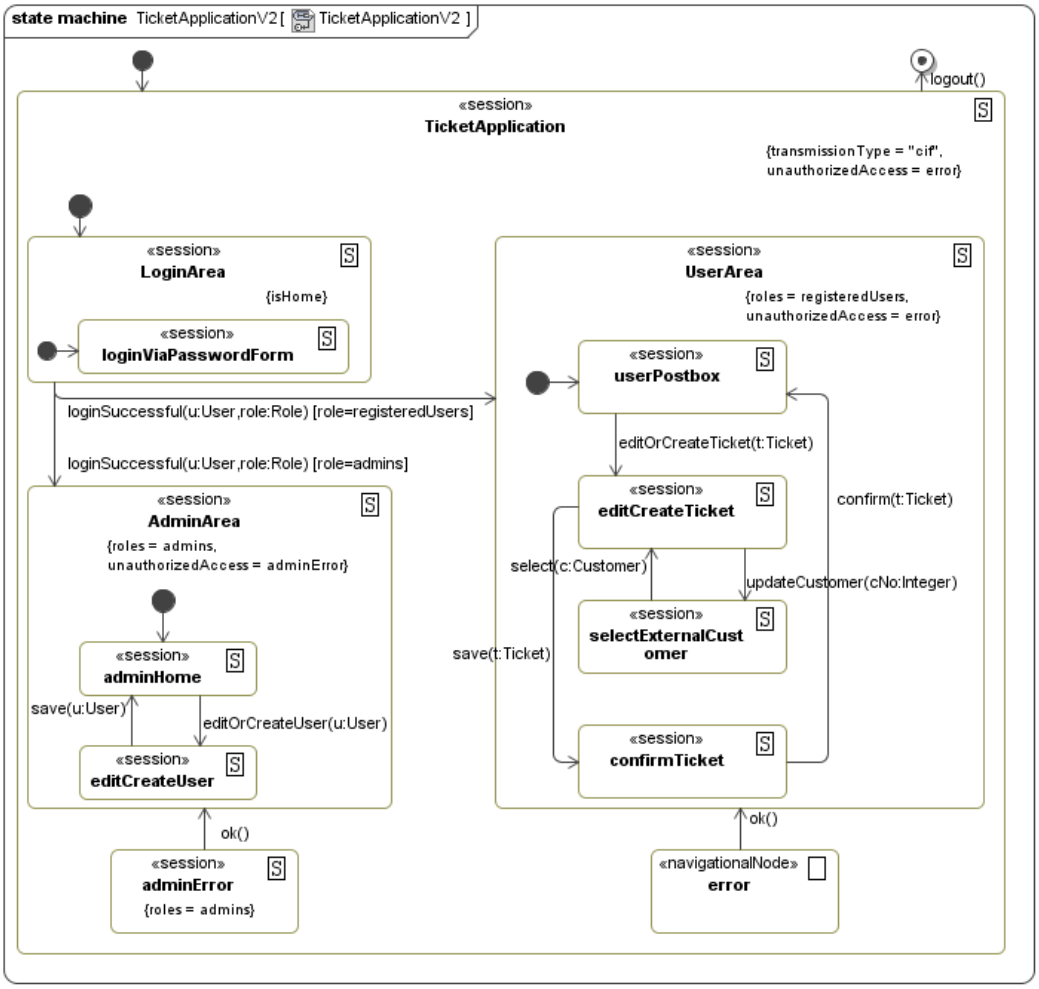

Figure 5.3: *TicketApplication: Navigation State Model*

The outermost state *TicketApplication* stands for the whole application. It holds the tag *unauthorizedAccess* containing the default violation node *error*. The tag *transmissionType="cif"* sets the overall type of data transmission during the session to `cif`, providing for confidentiality, integrity, and freshness: The implementation should prevent eavesdropping, replaying, or altering of transmitted data. [BKK11] As we can see, the navigational nodes, represented by the states on the innermost level, are grouped by three main areas or parent states: *LoginArea*, *AdminArea* and *UserArea*. Additionally, there are two different constraint violation nodes: *adminError* and *error*. In case of a constraint violation situation inside the *AdminArea* the user gets redirected to *adminError*. In any other constraint violation case the redirection points to *error*.

Every web user can access the login context node *loginViaPasswordForm* which is inside the *LoginArea* indicated by the *isHome* tag.

Logged in users with the role *admins* can access the whole *AdminArea*, containing the nodes *adminHome* and *editCreateUser*, indicated by the inherited roles tag. However, they must follow the deterministic SNP as defined by the transitions between the navigation states. This means to get access to *adminHome* the user must exhibit the role 'admins' and he must have been on *loginViaPasswordForm*, *adminError* or *editCreateUser* previously. In order to get access to *editCreateUser* the user needs to have the same role but must have been on the node *adminHome* right before. Otherwise the user gets redirected to the node *adminError* as described in the tag *unauthorizedAccess*.

Logged in users with the role *registeredUsers* are allowed to access the protected resources *userPostbox*, *editCreateTicket*, *selectExternalCustomer* and *confirmTicket* inside the *UserArea* by following the nondeterministic SNP as indicated by the transitions: *userPostbox* is the initial state. Therefore, every ticket-operation inside the *UserArea* must begin inside the *userPostbox* context. The next possible step is selecting a ticket or creating a new instance followed by the possibility to assign an external customer. Finally, the ticket has to be confirmed and the user gets back to the *userPostbox* context. The semantic meaning of the parent state *UserArea* can be interpreted in the same way.

5.4 Extracted Navigation File by *MagicSNP*

The navigation rule file depicted in Figure 5.4 is generated by *MagicSNP* dynamically corresponding to the state machine shown in Figure 5.3.

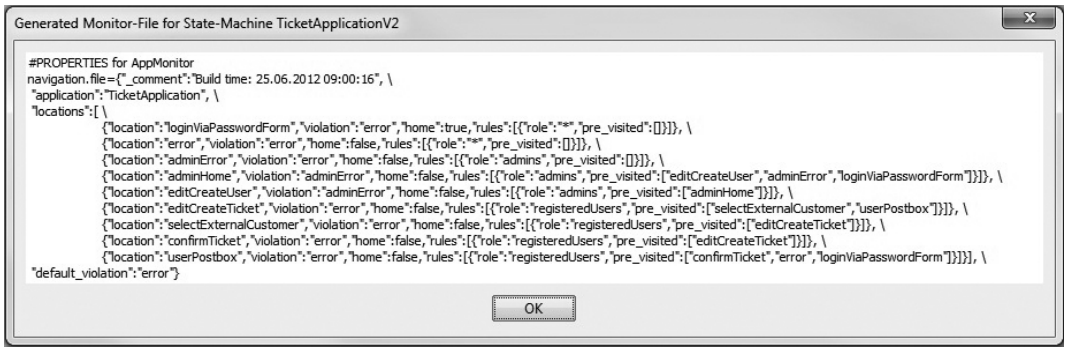

Figure 5.4: *JSON structured navigation rule file for TicketApplication*

It contains all the information needed for a monitor to provide navigational access control including SNPs for a web application: The default violation node is *error*, defined by the attribute *default_violation* on the outermost level. Furthermore, every single location entry holds the corresponding violation node and a list of access rules. Every single rule entry represents an user role which is allowed to access the current navigation node. In addition, the attribute *pre_visited* defines on which navigation nodes could the user have been right before to be inside a SNP.

5.5 Application of the Monitor Module

Following the security design phase the monitor module must be applied to the web application under consideration of these points:

i. The monitor module is accessible in the classpath of the web application

ii. The AppMonitor is registered as a filter on the URL-pattern `/protected/*` in the deployment descriptor

iii. The Java properties file `security.properties` is in the classpath of the web application containing the extracted navigation rule file

iv. Every view-node which is under access control has a directory named `protected` as last ancestor inside the application path structure

5.6 Used Frameworks and Technologies

For completeness, we provide the following list of the used frameworks and technologies for this sample web application:

i. Database: MySQL Server 5.5[1]

ii. Servlet container/Server: Apache Tomcat 7.0.8[2]

iii. Backend: Java EE 1.6[3], Spring Framework[4], Maven JSON-lib[5], Apache Commons[6]

iv. Frontend: Java Server Faces (JSF) 2.0 by Mojarra[7] and JavaServer pages Standard Tag Library (JSTL) 1.2[8], Openfaces[9]

v. Access Control: Our monitor module (applied as described in chapter 4)

5.7 Summary

The web application works properly as expected and according to the access control rules specified in the *Navigation State Model*. The main goal of providing access control under the consideration of SNPs is completely reached by this approach.

Summarizing, this case study shows the reliability of our whole approach and how easily it can be applied to a web application. The security based modeling part was straightforward and did not take much effort. At runtime the monitoring module does not cause latency issues at all as a result of using simple data structures and slim-efficient process cycles throughout the module. The clear error message handling contributes to the robustness and user-friendliness of the web application. In addition, the widespread usage of clear logging messages guarantees an adequate standard of maintainability for the system administrators.

[1] MySQL Server 5.5. `http://dev.mysql.com/downloads/mysql/`, last visited 2012-07-18
[2] Apache Tomcat 7.0.8. `http://tomcat.apache.org/download-70.cgi`, last visited 2012-07-24
[3] Java EE 1.6. `http://www.oracle.com/technetwork/java/javaee/downloads/index.html`, last visited 2012-07-08
[4] Spring Framework. `http://www.springsource.org/spring-framework/download`, last visited 2012-07-08
[5] Maven JSON-lib. `http://sourceforge.net/projects/json-lib/files/`, last visited 2012-07-03
[6] Apache Commons. `http://commons.apache.org/`, last visited 2012-07-10
[7] JSF 2.0. `http://javaserverfaces.java.net/download.html`, last visited 2012-07-18
[8] JSTL 1.2. `http://www.oracle.com/technetwork/java/index-jsp-135995.html`, last visited 2012-07-18
[9] Openfaces. `http://openfaces.org/downloads/`, last visited 2012-07-18

Chapter 6

Conclusion and Outlook

The goal of this thesis was to fill the gap of missing possibilities to model and control Secure Navigation Paths (SNPs) to enhance data protection within web applications. Therefore, we developed a new approach on how to model the aspect of SNPs for web applications efficiently using UWE's *Navigation State Model*. It turned out that compared to existing approaches which do not consider SNPs, our solution allows a more specific definition of navigational access control for web applications: Our modeling approach provides a modeling strategy which guarantees that every single user of the system has to stay on a limited number of paths in the intended order. First of all, this limits the possibilities of malicious attacks by safety-critical jumps through a web application. Additionally, this restriction allows to guide the user through a web application. Our new monitor module represents one possibility on how to apply the designed semantics of the modeling approach to a web application. Because of using simple data structures and efficient process cycles throughout the module, no latency issues arise. The link between our modeling solution and our monitor approach is given by our new MagicDraw plugin *MagicSNP*: It allows to validate the designed security model and to extract the corresponding navigation rules. Therefore, this plugin does what we expected: It secures and facilitates the handover between the modeling and the implementation phase of the software-development process. The applicability of our approaches is proven by our case study using a prototype called *Ticket Application*. Additionally, this case study demonstrates all fundamental advantages of our approaches: First, it shows how simple the navigational access control of an already existing web application can be modeled while considering user rules, violation behavior and SNPs. Second, it presents the way our monitor module and its components can be installed and configured without much effort for an already existing web application. Finally, this case study exhibits the comfort to validate the model and to extract all necessary access-control semantics with one simple mouse click using *MagicSNP*.

Future work might enhance the possibilities and dynamics of the cooperation between the security design phase and the application runtime. Based on the concepts of this thesis, a possible approach could be to link the design tool to a relational database which is also accessible by the monitor module. This would be an advantage, because at the moment we have to save the navigational rule file manually. By fetching the rules dynamically from a database, a restart of the application server could be avoided when loading new rule definitions. This would increase the efficiency of our approach. Our approach was tested for correctness, robustness, user-friendliness and applicability by a prototype web application. However, this prototype consists of nine different navigation nodes and two different user roles. In order to prove scalability we will develop

different web application prototypes which exhibit much more complexity.

To conclude, this thesis provides, to the extent of our knowledge, the only possibility to model and implement SNPs. Since SNPs raise the security of access control on a higher level, the presented approaches should be concerned within the context of data protection in modern web applications.

List of Figures

Acronyms

API	Application Programming Interface
CASE	Computer-Aided Software Engineering
CD	Compact Disc
CSS	Cascading Style Sheets
DI	Dependency Injection
GUI	Graphical User Interface
IDE	Integrated Development Environment
JAR	Java Archive
JDBC	Java Database Connectivity
JSF	Java Server Faces
JSON	JavaScript Object Notation
JSTL	JavaServer pages Standard Tag Library
LDAP	Lightweight Directory Access Protocol
OCL	Object Constraint Language
PDF	Portable Document Format
RBAC	Role-Based Access Control
SNP	Secure Navigation Path
UML	Unified Modeling Language
URL	Uniform Resource Locator
UWE	UML-based Web Engineering

Content of the CD

The content of the enclosed CD is organized as follows:

- `/`
  - `Implementation` ..... Sources of implementations introduced in this thesis
    - `MagicSNP` ..... Implementation of *MagicSNP* with Java
    - `SNPMonitor` ..... Implementation of our monitoring module with Java
    - `TicketApplication` ..... Implementation of our case-study with Java and JSF
  - `Metadata` ..... IDE-configuration instructions, *TicketApplication* MySQL-dump, timetable
  - `Paper` ..... Copies of the related work that is referenced in the thesis
  - `Thesis` ..... The written thesis in LaTeX and PDF format
    - `Chapters` ..... Chapters as tex-files
    - `Images` ..... Images used in the thesis
    - `Presentation` ..... The *Oberseminar* presentation

Bibliography

[And08] R.J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd edn.* Wiley, Chichester, 2008.

[BB78] B. Boehm and J. Barry. *Characteristics of software quality.* North-Holland Pub. Co., 1978.

[BCE11] D. Basin, M. Clavel, and M. Egea. A decade of model-driven security. *Ruth Breu, Jason Crampton, and Jorge Lobo*, pages 1–10, 2011.

[BK09] M. Busch and N. Koch. MagicUWE - A CASE Tool Plugin for Modeling Web Applications. *Gaedke, M., Grossniklaus, M., Diaz, O. (eds.) ICWE 2009. LNCS*, 5648:505–508, 2009.

[BKK11] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. *Janis Grabis and Marite Kirikova, editors, 10th International Conference on Business Perspectives in Informatics Research*, 2011.

[Jür04] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[KK11] N. Koch and A. Kraus. The Expressive Power of UML-based Web Engineering. *IWWOST´2002*, 2011.

[LBD02] T. Lodderstedt, D. Basin, and J. Doser. A UML-Based Modeling Language for Model-Driven Security. *Proceedings of 5th International Conference on the Unified Modeling Language*, 2460:426–441, 2002.