Problem Frames and Architectures for Security Problems

Denis Hatebur[1] and Maritta Heisel[2]

[1] Universität Duisburg-Essen, denis.hatebur@uni-duisburg-essen.de and Institut für technische Systeme GmbH, d.hatebur@itesys.de
[2] Universität Duisburg-Essen, Germany, Fachbereich Ingenieurwissenschaften, maritta.heisel@uni-duisburg-essen.de

Abstract. We present several problem frames that serve to structure, characterize and analyze software development problems in the area of software and system security. These problem frames constitute *patterns* for representing security problems, variants of which occur frequently in practice. Solving such problems starts with the development of an appropriate software architecture. To support that process, we furthermore present architectural patterns associated with the problem frames. We illustrate our approach by the example of an electronic purse card.

1 Introduction

Problem frames were developed by Michael Jackson [6]. He describes them as follows (emphasis ours): "A problem frame is a kind of *pattern*. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."

Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure. Patterns are defined for different activities at different stages of the software life cycle. *Problem Frames* [6] are patterns that classify software development *problems*. *Architectural styles* are patterns that characterize software architectures [1, 11]. They are also called "architectural patterns" (see Section 2.2). *Design Patterns* [5] are used for finer-grained software design[3], while *idioms* are low-level patterns related to specific programming languages [3].

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time. The problem frames defined by Jackson cover a large number of software development problems, because they are quite general in nature. To support software development in more specific areas, however, specialized problem frames are needed.

In this paper, we present four problem frames that capture software development problems occurring frequently in the area of software and system security. We call these problem frames *security frames*. Two of our security frames concern authentication. The third one deals with the secure (i.e., encrypted) transmission of data, and the fourth one is suitable for generating and storing security information (such as public and private keys, PINs).

[3] Design patterns for security have also been defined, see Section 5.

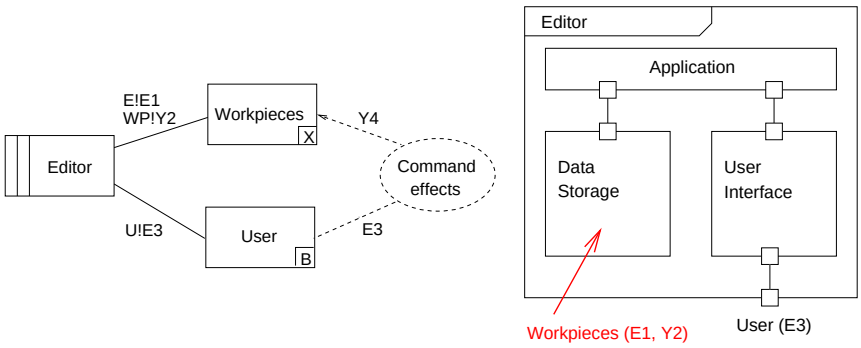

Fig. 1. Workpieces Frame Diagram and Architectural Pattern

Architectural patterns are suitable solution structures for problem frames, because architectural design is one of the first activities in solving software development problems. Hence, the gap between the problem description and the software architecture is not too large, and we can establish direct relations between problem structures and solution structures. As we have shown in [4], one can define architectural patterns that reflect the characteristics of the different problem frames. In much the same way, we equip our security frames with corresponding architectural patterns.

Section 2 describes the basics of our work, while the security frames and corresponding architectures are presented in Section 3. We illustrate our approach by developing a secure electronic purse card in Section 4. Section 5 discusses related work, and we conclude in Section 6.

2 Problem Frames and Architectural Patterns

In this paper, we present new problem frames for security problems and the corresponding architectural patterns. As a notation for our architectural patterns, we use composite structure diagrams of UML 2.0 [12]. In the following, we give brief descriptions of these basic concepts of our work.

2.1 Problem Frames

Problem frames are described by *frame diagrams*, which basically consist of rectangles and links between these, see left-hand side of Fig. 1. The task is to construct a *machine* that improves the behavior of the environment it is integrated in.

Plain rectangles denote application domains (that already exist), a rectangle with a double vertical stripe denotes the machine to be developed, and requirements are denoted with a dashed oval. The connecting lines represent interfaces that consist of shared *phenomena*. A dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

Jackson distinguishes *causal* domains that comply with some laws, *lexical* domains that are data representations, and *biddable* domains that are usually people. Jackson defines five basic problem frames (*Required Behaviour, Commanded Behaviour, Information Display, Workpieces* and *Transformation*). As an example, we present the *Workpieces* frame in more detail. The following problems fit to that problem frame [6]: “A

tool is needed to allow a user to create and edit a certain class of computer processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analyzed or used in other ways. The problem is to build a machine that can act as this tool." The "X" indicates that the *Workpieces* domain of the frame diagram shown on the left-hand side of Fig. 1 is a lexical domain. The notation "U!E3" means that the user commands *E3* are controlled by the (biddable) *User* domain. Similarly, the phenomena *E1* are the commands used by the *Editor* to change the *Workpieces* domain. The shared phenomena *Y2* represent the state of a workpiece; they are controlled by the *Workpieces* domain. The shared phenomena *Y4* need not be the same as *Y2*. They will often have some meaning to the user, whereas the phenomena *Y2* are phenomena accessible by the machine.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements, and it is not shown who is in control of the shared phenomena. An example of a context diagram is shown in Fig. 8. Then, the problem is decomposed into subproblems. If ever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, phenomena, interfaces and requirements. The instantiated frame diagram is called a *problem diagram* (for an example, see Fig. 9). It describes the problem as a whole. Since the requirements refer to the environment in which the machine must operate, the next step consists in deriving a *specification* for the machine, using domain knowledge. In that process, non-implementable requirements are transformed into implementable ones. (For a more detailed description, see [7].) The specification is the starting point for the development of the machine.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. Since all problems fitting in a problem frame share the same characteristic properties, their solutions will have common characteristic properties, too. Therefore, it is worthwhile to look for solution structures that match the problem structures defined by problem frames.

2.2 Architectural Styles

According to Bass, Clements, and Kazman [1], "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 the relationships among them." Architectural styles are patterns for software architectures.

When choosing an architecture for a system, usually several architectural styles are possible. However, instead of considering *all* possible architectures, we propose *specific* architectural patterns for our security frames in order to provide a concrete starting point for the further development of the machine. The architectural patterns we have defined for Jackson's problem frames (see [4]) and the ones we will define for security frames are based on a *layered architecture*. The components in this layered architecture are either *communicating processes* (active components), or they are used with a *call-and-*

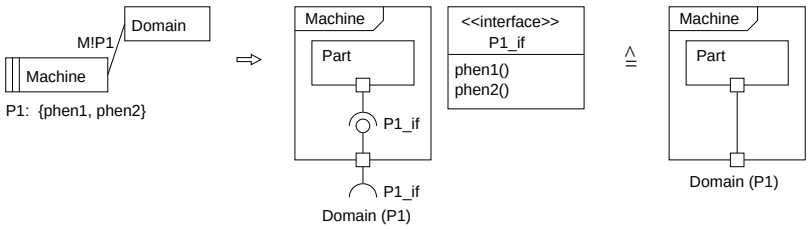

Fig. 2. Notation for Architectures

return mechanism (passive components). That design decision is taken in a later step of the development. In [4], we also show how the *repository* and the *pipe-and-filter* architectural styles can be integrated into the layered architecture. We use UML 2.0 composite structure diagrams (see Section 2.3) to represent architectural patterns as well as concrete architectures.

The architectural pattern shown on the right-hand side of Fig. 1 contains a user interface component, because the problem frame diagram contains a user. The data storage component of the architecture corresponds to the *Workpieces* domain of the frame diagram. The *Editor Application* component is responsible for manipulating the data storage according to the user commands. Note that there is only one interface with the environment – namely the interface with the user – because the lexical *Workpieces* domain is part of the machine.

2.3 Composite Structure Diagrams

Composite structure diagrams [12] are a means to describe architectures. They contain named rectangles, called *parts*. These parts are components of the software. Each component may contain other (sub-) components. Atomic components can be described by state machines and operations for accessing internal data. In our architectures, components for data storage are only included if the data are stored persistently. Otherwise they are assumed to be part of some other component. Parts may have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the “lollipop” notation, and required interfaces using the “socket” notation.

Fig. 2 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams. The partial problem diagram shown on the left-hand side of Fig. 2 states that the phenomena *phen1* and *phen2* shared between the machine and a domain are controlled by the machine. In the composite structure diagram (with associated interface class) shown in the middle of Fig. 2, this is expressed by a required interface *P1_if* of the *part* component of the machine, which is the same as for the whole machine. Shared phenomena controlled by a domain correspond to provided instead of required interfaces of the *part* and the machine, respectively. Because of this direct correspondence, we do not use the socket and lollipop notation in the following, but use connectors between ports, as shown on the right-hand side of Fig. 2. These connectors can be implemented e.g. as data streams, function calls, asynchronous messages or hardware access.

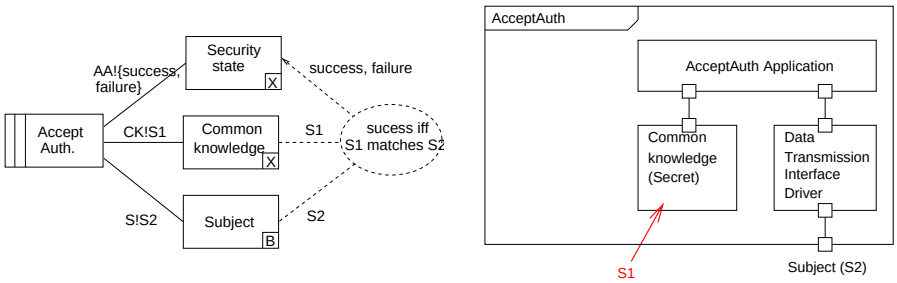

Fig. 3. Accept Authentication Information Frame Diagram and Architectural Pattern

3 Security Frames and Architectural Patterns

We now present the four security frames we have developed, together with the corresponding architectural patterns that define structures for the machine domains of the security frames.

The first two security frames are concerned with authentication. We distinguish two authentication frames. In the first frame, a subject must authenticate itself to the machine to be constructed. In the second frame, the machine to be constructed must authenticate itself to some other subject. The third security frame deals with the secure transmission of data over an insecure channel, and the fourth frame is applicable when common security knowledge must be distributed with the help of a trust center. None of these problem classes is addressed by Jackson's problem frames.

3.1 Accept Authentication Frame

For security systems, authentication of users and other components is an important concern. Authentication is necessary to allow access to some other information. That information is not part of the problem and hence not part of the frame diagram shown on the left-hand side of Fig. 3.

The *Subject* in the frame diagram can be a user or another machine. To make authentication possible, there must be a common knowledge between subject to be authenticated and the machine. If the authentication information *S2* provided by the subject matches the common knowledge *S1* stored in the machine, then the authentication is successful. Otherwise, it fails. That information is represented by the domain *Security state*.

In the corresponding architectural pattern on the right-hand side of Fig. 3, we include the *Common knowledge*, but we do not include the *Security state* because it should not be stored persistently and hence does not correspond to an architectural component. Instead, it is reflected in the internal state of the part *AcceptAuth Application* that is responsible for enabling or disabling other functionality.

3.2 Submit Authentication Frame

Because the subject might be a system, there exists the problem *Submit Authentication*. The frame diagram and the corresponding architectural pattern are shown in Fig.

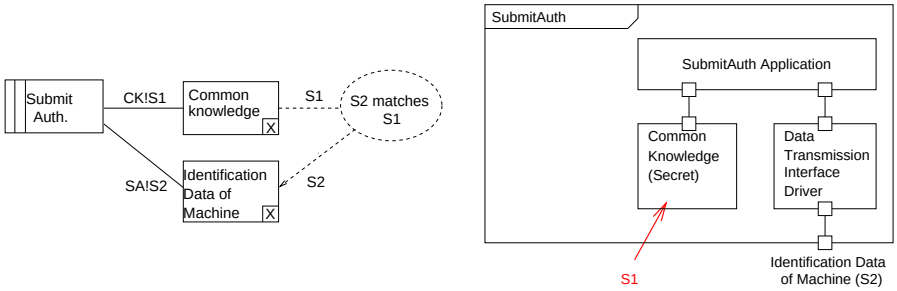

Fig. 4. Submit Authentication Information Frame Diagram and Architectural Pattern

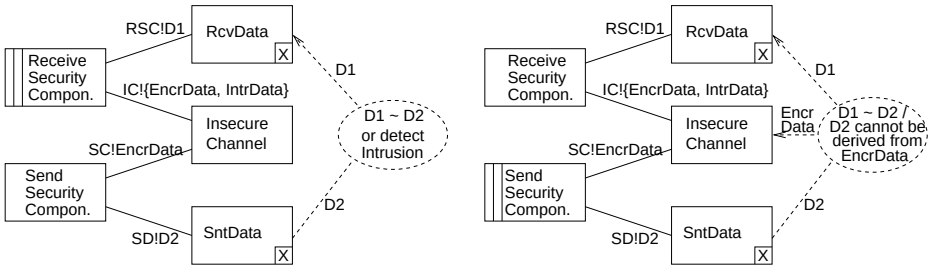

Fig. 5. Secure Data Transmission Frame Diagrams

4. For this problem, the *Security state* is part of the subject to which the machine to be built wants to authenticate itself. Therefore the *Security State* it is not part of the frame diagram. The machine has to use the *Common knowledge* to generate matching *Identification Data* for the subject.

3.3 Secure Data Transmission Frames

Another important security problem is the secure transmission of data. We need to build a security component that receives data from another component or sends data to another component over an insecure channel. That situation is depicted in Fig. 5 on the left-hand side for receiving data and on the right-hand side for sending data.

The security component at the bottom of the figure wants to send data (domain *SntData*, phenomena *D2*) to the security component at the top of the figure. Because the transmission channel is insecure, the data is encrypted (phenomena *EncrData*). It is possible that the insecure channel transmits some intruder data *IntrData* instead of the original encrypted data. The encrypted data or intruder data will be decrypted by the security component shown on the top of the figure, yielding *D1*. The requirement for receiving data states that either the data *D1* and *D2* match, or the intrusion will be detected (integrity). The requirement for sending data states that the data *D1* and *D2* match, and that the *D1* cannot be derived from *EncrData* (integrity and confidentiality).

For this class of problems, we propose the architectural patterns shown in Fig. 6. In this architecture, a storage for a secret is necessary in addition to the data storage. If this storage is persistent it is an additional part in the architecture. Otherwise, the storage component is not included, as indicated by the notation [0..1].

3.4 Distribute Security Information Frame

For the architecture shown in Fig. 6, it is necessary that each machine has some common knowledge. This raises the problem of how to distribute that common knowledge. Fig. 7 shows the frame diagram. The common (secret) knowledge is transferred to the machine by a trusted component, the *Trust Center*, over a secure channel. The requirement states that indeed the correct common knowledge is stored in the machine.

The corresponding architectural pattern contains a part *ManageSecretApplication* that has to store the secret (or common knowledge) and restrict the access to it. Its purpose is to manage the secret.

4 Case Study: Electronic Purse Card

The illustrate the usage of security frames, we consider a smart card with a simplified electronic purse application using asymmetric encryption. This smart card is used to ensure secure payment. To pay with the card, the user has to enter a PIN at a card reader. The authorization of the card is checked via a website. The card also has to check the authorization of the website. The transmitted data have to be protected against unauthorized read and change access. To allow payment, money must be loaded on the card. This is only possible if the the account information allows this transaction (the card can be locked).

4.1 Requirements and Context Diagram

The following requirements must be met. We number them in order to reference them in the description of the different subproblems.

R1 Loading money on the card is possible if the account information allows to do this transaction.
R2 Paying with the card is possible if there is enough money on the card.
R3 Authentication of the card is necessary for paying and loading money.
R4 Authentication of the website is necessary for paying and loading money.
R5 Authentication of the user using a PIN is necessary for paying.

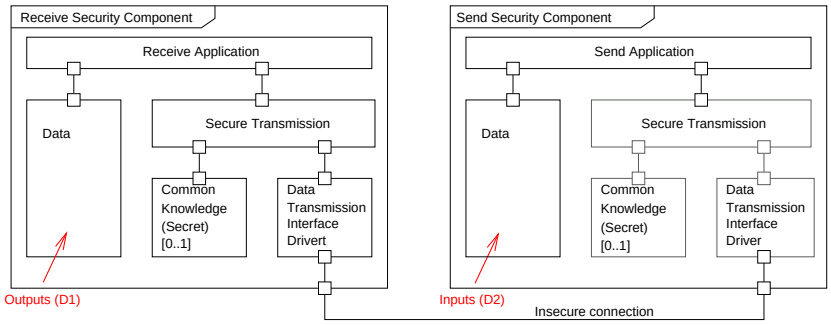

Fig. 6. Secure Data Transmission Architectural Pattern

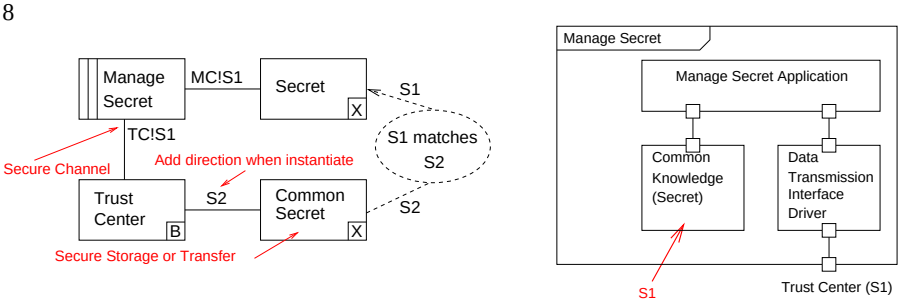

Fig. 7. Distribute Security Information Frame Diagram and Architectural Pattern

R6 The card should prevent replay-intrusion and even prevent somebody else from reading transmitted information (man-in-the-middle attack).

R7 It should not be possible to copy the card.

R8 Only a card and a website personalized by a trust center should be usable for transactions.

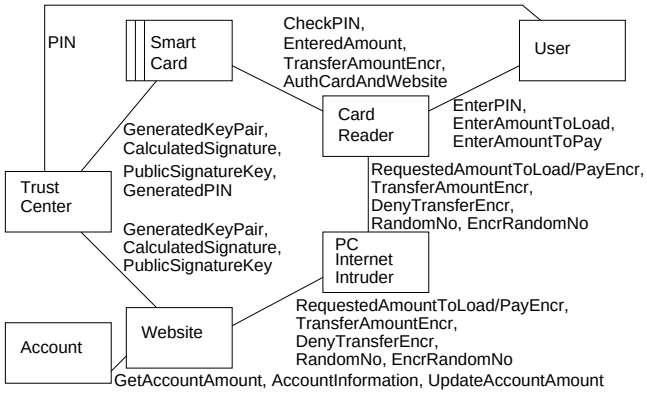

Fig. 8. Context diagram for Electronic Purse Card

Fig. 8 shows the context diagram corresponding to this problem. It contains the relevant domains and shared phenomena. The domains *SmartCard*, *Website* and *Account* occur only once in the diagram. However, the system will work with different instances of these domains. It will be able to handle different smart cards, and it will be connected via different websites to different accounts. Moreover, the interface between the *CardReader* and the *User* has been simplified. The domain *PC, Internet, Intruder* denotes the insecure channel that involves a PC, the Internet, and possibly an intruder.

The following table shows the subproblems that can be identified, the problem/security frame the subproblem fits to, and the requirements that are covered. In the following, we present one instantiation for each of the introduced problem/security frames.

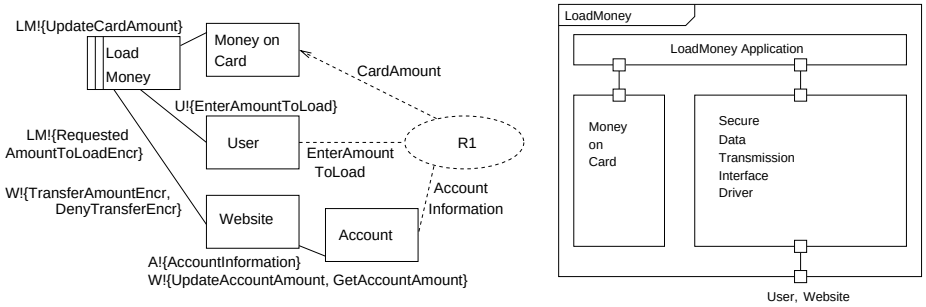

Fig. 9. Load Money Subproblem Diagram and Architecture

Subproblem	Frame	Reqs.
Load Money	Workpieces	R1
Pay	Workpieces	R2
Authenticate Card	Submit Authentication	R3
Authenticate Website	Accept Authentication	R4
PIN Authentication	Accept Authentication	R5
Receive Secure Data	Secure Data Transmission	R6
Send Secure Data	Secure Data Transmission	R6
Distribute Keys	Distribute Security Information	R7, R8
Distribute PIN	Distribute Security Information	R7, R8

4.2 Subproblem: Load money

This subproblem is concerned with loading money on the card (R1). It fits to a variant of Jackson's *Workpieces* problem frame. It is extended with the constraint that only if the account information allows it, the amount of money can be loaded onto card. The problem shown in Fig. 9 states that the *CardAmount* should change according to *EnterAmountToLoad* and *AccountInformation*.

The problem diagram of Fig. 9 is derived from the context diagram of Fig. 8 as follows: the domain *Trust Center* is not relevant for this subproblem. The connection domains[4] *Card Reader* and *PC, Internet, Intruder* are left out in this subproblem, because the connection is assumed to be secure, the security of the connection being covered by other subproblems. To describe this problem, we split the domain *SmartCard* into *Money on Card* and *Load Money*.

The problem of Fig. 9 shows the requirements the machine must achieve when integrated into its environment. As noted earlier, the requirements must be transformed into a specification that describes the behavior of the machine. In the area of security, protocols [9] exist that make it possible to transform requirements such as "secure transmission" of "authentication" into sequences of messages exchanged between different partners.

[4] These are domains that serve to connect two other domains. If a connection domain is reliable and does not cause significant delays, it may be ignored, see [6].

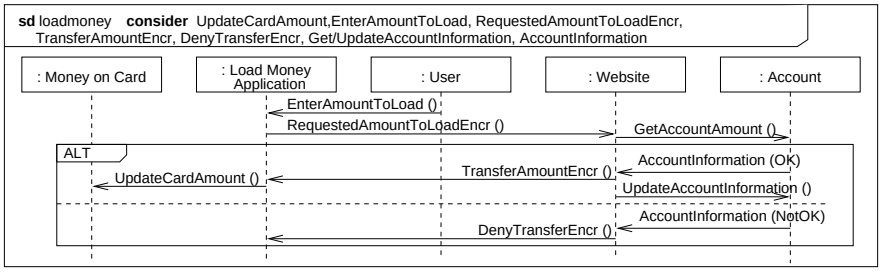

Fig. 10. Load Money Sequence Diagram

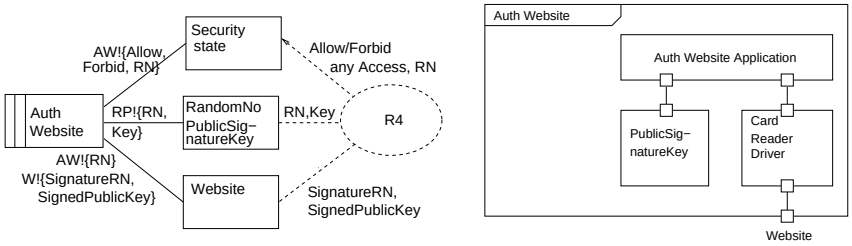

Fig. 11. Website Authentication Subproblem Diagram and Architecture

Fig. 10 shows a UML 2.0 sequence diagram that represents the specification of the machine *Load Money*. When the *User* enters the amount of money (*EnterAmountToLoad*) the message *RequestedAmountToLoadEncr* will be sent to the *Website*. The *Website* will check the *AccountInformation*. The sequence diagram in Fig. 10 describes the following two alternatives, marked with the keyword *ALT*. If the *AccountInformation* allows to load the requested amount of money on card, the amount of money on the *Account* will be updated (*UpdateAccountInformation*), the amount will be transmitted (*TransferAmountEncr*), and *Money on Card* will be updated (*UpdateCardAmount*[5]). Otherwise the phenomenon *DenyTransferEncr* occurs. For reasons of space, we do not give the sequence diagrams for the other subproblems. For this subproblem, the *Website* is the website of the user's bank, and the *Account* is the user's account, which is debited with the account loaded onto the card.

The right-hand side of Fig. 9 shows the corresponding architecture, which is an instantiation of the pattern given in Fig. 1. Here, *User* and *Website* are connected to the machine via a *Secure Data Transmission Interface*.

4.3 Subproblem: Authenticate Website

An authentication of the website is required in R4. Here, we instantiate the "Accept Authentication" frame as shown in Fig. 11. For this authentication, a *Random Number* should be used to prevent replay intrusion. Therefore, we need to add the random number RN as a shared phenomenon between the *Auth Website* machine and *Website*, controlled by the machine.

[5] With the new domain *Money on Card* the shared phenomenon *UpdateCardAmount* has to be added.

The part *Auth Website* of the architecture shown in Fig. 11 must contain a part *RandomNo PublicSignatureKey* that can generate random numbers with sufficient quality. To check the authenticity of the website, the website encrypts the random number provided by the card with its private key (it generates the signature of the random number (*SignatureRN*). This signature can be checked using the public key of the website. To make it possible that new websites can be added to the system without replacing all cards, the website has to provide its own public key. The card can check the provided public key using a signature of a Trust Center. The interaction for a combination of submitting and accepting authentications can be found in [10].

4.4 Subproblem: Receive Secure Data

To prevent replay intrusion and read access on transmitted data, we define the subproblem shown in Fig. 12. The *Card Reader* and the *Trust Center* are not directly relevant for this subproblem. Moreover, is not relevant what data are transmitted. Therefore we take an abstraction from *Load Money* and *Pay*. Also the messages *TransmitAmountEncrMessage* and *CheckAmountEncrMessage* are merged to *AccessAmountEncr*.

A common secret as described in the architectural pattern of Fig. 6 is not stored persistently on the card. It can be derived from the random number and can be changed for each transmission. Hence, the architecture of Fig. 12 (derived from the pattern given in Fig. 6) does not contain a corresponding component.

4.5 Subproblem: Distribute Keys

Requirements R7 and R8 express that only the trust center may generate a valid card. Important for a valid card are the PIN and the keys. In this subproblem, we focus on the keys. The requirements are covered partly in the subproblem shown in Fig. 13, where the *Common Secret* domain that is part of the *Trust Center* is shown separately. That subproblem is an instance of the “Distribute Security Information” frame.

The trust center has to generate an individual public/private key pair for each card and write it onto the card. To guarantee that this key pair is valid and originated from the trust center, it is signed with the private key of the trust center. To allow the card to authenticate other systems, it needs the public key of the trust center. This also is written onto the card.

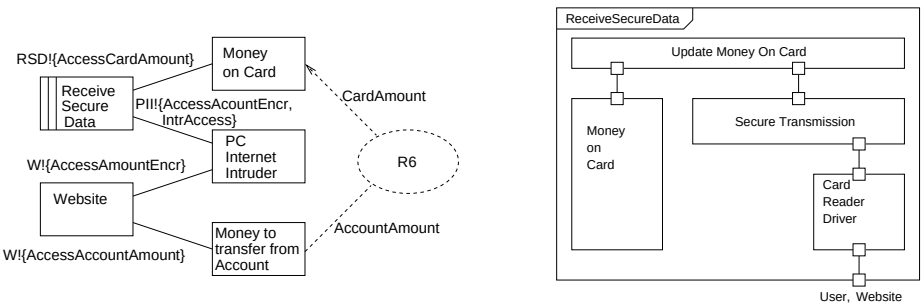

Fig. 12. Receive Secure Data Subproblem Diagram and Architecture

The subproblem the machine *Manage Secrets* has to solve is to manage the access to the security information. After producing the card, everybody owning an uninitialized card can initialize it. But only the trust center is able to generate a signature with its private key that allows the key to be used in the payment system. The machine has to manage the access to the secrets. After initializing the card, the functionality to change the security information is disabled. The private part of the key pair must also be protected against read access. Moreover, all other functionality has to be disabled as long as the card is not yet initialized.

The architecture of Fig. 13 is an instantiation of the pattern given in Fig. 7.

4.6 Composed Architecture

We now must compose the architectures developed for the subproblems to obtain an architecture for the whole Smart Card. For doing this, we must find the parts occurring in different subproblem architectures that must be mapped to the same component in the composed architecture.

The composed architecture for the Smart Card is shown in Fig. 14. The component *Amount Of Money* combines the persistent storage of the subproblems *Pay* and *Load Money*. The *Load Money/Pay-Application* combines the behavior of the machines *Load Money* and *Pay*. The component *Secure Data Transmission Interface* is replaced by the components of the security subproblem architectures (including the ones given in Figs. 11–13). The *Card Reader Driver* is the same in all security subproblem architectures and can be used directly in the composed architecture. The functionality of the remaining two components have to be derived from those in the security subproblem architectures.

For all components, their exact specifications must be set up, and it must be shown that the components work together in such a way that they fulfill the specifications of all machines corresponding to the different subproblems. The functionalities of the different architectural parts are now clear, as well as the interfaces between them. Thus, we have established an appropriate starting point for the further development of the smart card system in a systematic way.

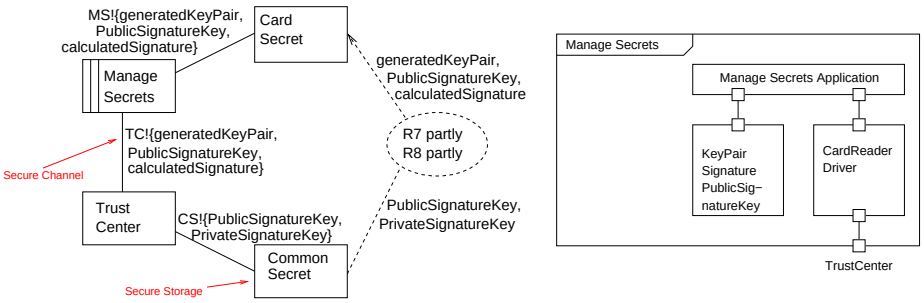

Fig. 13. Distribute Keys Subproblem Diagram and Architecture

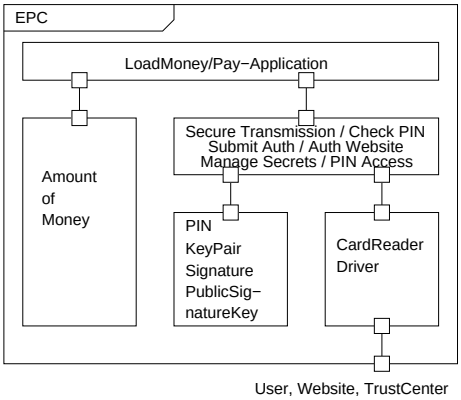

Fig. 14. Composed Architecture

5 Related Work

Our security frames are related to abuse frames on the one hand and to security patterns on the other hand.

Security frames treat security requirements in the same way as other (functional) requirements, and the goal is to construct a machine that fulfills the security requirements. Lin et al. [8] take another approach to use the ideas underlying problem frames in security. They define so-called anti-requirements and the corresponding abuse frames. An anti-requirement expresses the intentions of a malicious user, and an abuse frame represents a security threat. The purpose of anti-requirements and abuse frames is to analyze security threats and derive security requirements. Thus, the two approaches complement each other. Abuse frames can be used to derive the security requirements that can then be addressed with security frames.

While abuse frames can be used earlier in the software development process than security frames, security patterns [2] are applied in a later phase, namely the phase of detailed design. The relation between security frames and security patterns is much the same as the relation between problem frames and design patterns: the frames describe problems, whereas the design/security patterns describe solutions on a fairly detailed level of abstraction. Moreover, design and security patterns are applicable only in an object-oriented setting, while problem and security frames are independent of a particular programming paradigm.

6 Conclusion

In this paper, we have presented a new kind of problem frames tailored for representing security problems, called security frames. Security frames are patterns for software development problems occurring frequently when security-critical software has to be developed.

The security frames presented in this paper are intended to be the first in a more complete collection. Once a (relatively) complete collection of security frames is defined, it is of considerable help for developers. For a new security-critical system to be

constructed, the security frame catalogue can be inspected in order to find the frames that apply for the given problem. Thus, a security frame catalogue helps to avoid omissions and to cover all security aspects that are relevant for the given problem.

Furthermore, the security frames help to decompose complex security problems to simpler ones that can be handled by standard mechanisms. Like design and security patterns, security frames can establish a common vocabulary and shared knowledge between developers of security-critical systems.

While the security frames themselves "only" help to comprehend, locate and represent problems, our architectural patterns associated with the different security frames propose concrete structures for *solving* the problems fitted to security frames. The architectural patterns also help to compose the solutions of the different subproblems in order to construct the complete system, as is shown in more detail in [4].

With the concept of security frames and corresponding architectural patterns (in addition to abuse frames and security patterns), one can hope to cover large parts the development of security-critical software with a pattern-based approach.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] B. Blakley and C. Heath. *Technical Guide: Security Design Patterns*. The Open Group, April 2004. `http://www.opengroup.org/publications/catalog/g031.htm`.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] C. Choppy, D. Hatebur, and M. Heisel. Architectural patterns for problem frames. *IEE Proceedings – Software, Special issue on Relating Software Requirements and Architecture*, 2005. To appear.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[6] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[7] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[8] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings of 11th IEEE International Requirements Engineering Conference (RE'03)*, pages 371–372, 2003. Poster Paper.

[9] C. P. Pfleeger. *Security in Computing*. Prentice Hall, 1996.

[10] T. Rottke, D. Hatebur, M. Heisel, and M. Heiner. A problem-oriented approach to common criteria certification. In S. Anderson, S. Bologna, and M. Felici, editors, *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 2434, pages 334–346. Springer-Verlag, 2002.

[11] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[12] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.