Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design

Holger Schmidt and Ina Wentzlaff

University Duisburg-Essen, Faculty of Engineering, Department of Computer Science, Workgroup Software Engineering, Germany

{holger.schmidt, ina.wentzlaff}@uni-duisburg-essen.de

Abstract. In this paper, we present a pattern-based software development method that preserves usability and security quality characteristics using a role-driven mapping of requirements analysis documents to architectural design artifacts. The quality characteristics usability and security are captured using specialized problem frames, which are patterns that serve to structure, characterize, and analyze a given software development problem. Each problem frame is equipped with a set of appropriate architectural styles and design patterns reflecting usability and security aspects. Instances of these architectural patterns constitute solutions of the initially given software development problem. We illustrate our approach by the example of a chat system.

1 Introduction

Besides the functional aspects of a software system, a software engineer must face *quality characteristics* such as security and usability. In general, all software systems have quality requirements, even if they are often acquired insufficiently and less considered compared to functional aspects during the software development life cycle. Causing serious damage to the economy (e.g., a stock market system, market share, and sales market of software product), endangering personal privacy or threatening people's life (e.g., a medical chip card system, traffic accidents, or airplane disasters) can be possible consequences if software neglects usability needs or security demands. Many security-critical software systems fail because their designers protected the wrong things, or protect the right things but in the wrong way. Inadequate usability is a reason for user activities causing undesired and dangerous software system effects. Thus, adequate security and usability engineering requires to have an explicit understanding of the security and usability requirements and to provide effective techniques to accomplish them.

Knowing that building systems with security and usability demands is a highly sensitive process, it is important to reuse the experience of commonly encountered challenges in these fields. This idea of using *patterns* has proved to be of value in software engineering for years, and it is also a promising approach in security and usability engineering. 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 or behavior. Patterns are defined for different activities at different stages of the software development lifecycle. *Problem frames* [9] are patterns that classify software development *problems*. *Architectural styles* are patterns that

characterize software architectures [2]. In Software Engineering *design patterns* [6] are commonly used for finer-grained software design and they are as well used for coarse-grained architectural design.

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 [9] cover a large number of software development problems, because they are quite general in nature. To support software development in more specific areas such as security and usability engineering, however, *HCI-oriented problem frames (HCIFrames)* [18] have been developed for usability engineering, while *security problem frames* and *concretized security problem frames* [8] have been developed for security engineering.

In this paper, we show how to use the problem frames approach in the area of security and usability engineering to develop architectures. We propose a pattern-based method developed for *preserving* security and usability characteristics from requirements analysis to architectural design. Section 2 gives an overview of this method. Initially, a software engineer must understand the context of a software development problem and decompose the overall problem situation into smaller subproblems (Sections 2.1 and 2.2). For this purpose, we apply problem frames defined by Jackson [9] and specialized problem frames for security and usability demands (Sect. 3). To preserve the quality characteristics identified and collected using specialized problem frames, we equip each problem frame with corresponding architectural patterns. Then, entities, facets, and their interactions in the problem description represented within the instantiated problem frames are mapped by a role-driven process to corresponding components and classes of the solution description (Sect. 4).

Additionally, security and usability problem frames can be systematically transformed into notations of the Unified Modeling Language (UML) [17]. This increases their value in later software development phases. Thus, we obtain a software design based on commonly known architectural patterns and achieve a seamless transition from requirements analysis to software design, preserving quality characteristics. We illustrate our approach by developing a chat system, and conclude our in Sect. 5.

2 A Pattern-Based Software Development Approach

We propose a pattern-based software development process consisting of four steps, which will be described in detail in the following sections:

1. Understand the problem situation (Sect. 2.1)
2. Decompose overall problem into simple subproblems (Sect. 2.2)
3. Fit subproblems to problem frames (Sect. 3.1)
   (a) Identify quality characteristics (Sect. 3.2)
   (b) Classify subproblems according to quality demands (Sect. 3.3)
4. Instantiate corresponding architectural and design patterns (Sect. 4)

We illustrate our approach by the pattern-based development of a chat application, starting from the requirements analysis and leading to the derivation of software design artifacts.

The starting point for the analysis of our software development project (the *system mission*) can be outlined in one simple sentence:

> *"A text-message-based communication platform shall be developed which allows multi-user communication via private I/O-devices."*

Requirements describe the application environment when our developed software is in operation. They represent desired properties of the problem domain. In contrast, *domain knowledge* describes given properties of the environment (*facts*) and important environmental conditions (*assumptions*). Desired and given properties of the problem domain are summarized by a context diagram. It describes the overall problem situation, which we want to improve through our software product.

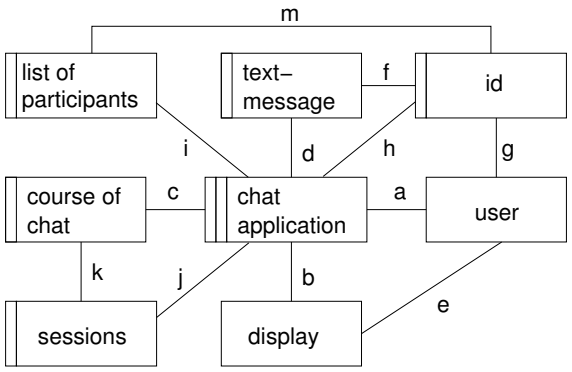
In Tab. 1 requirements `R1` - `R10` and domain knowledge (consisting of some facts `F1` - `F2` and assumptions `A1` - `A2`) are collected to elaborate our system mission.

`R1`	Users can phrase text-messages, which are shown on their private graphical displays.
`R2`	Users send their phrased text-messages to the chat, which are stored in the public course of the chat in their correct temporal order.
`R3`	The course of the chat is shown to the users on their private graphical displays.
`R4`	Sending text-messages changes the presentation of the course of the chat on the user's graphical displays.
`R5`	Each text-message is related to its respective user, so that the originator of a message can be identified.
`R6`	All users are stored in a list of participants, which is visible to every chat user.
`R7`	To each course of the chat a corresponding list of participants is shown.
`R8`	Various chat sessions considering different subjects of discussion are offered to the users.
`R9`	All available chat sessions are shown to the users.
`R10`	Users can switch among different chat sessions.
`F1`	Users can only understand the course of the chat, if the text-messages are presented in their correct temporal order (First In - First Out (*FIFO*)).
`F2`	If more than two users participate in the chat, it is required to relate messages to their originators in order to maintain a comprehensible chat communication.
`A1`	Users will follow the course of the chat on their private graphical display.
`A2`	Several users will participate in the chat.

Table 1. Requirements and domain knowledge for the chat application

2.1 Understand the Problem Situation

In the terminology of Jackson [9], the software development goal is to build a *machine* that changes the environment in a specified way. Thus, an intensive investigation of the given and desired properties of the problem environment is mandatory (cf. Tab. 1). This requirements and domain analysis process is accompanied using a *context diagram* that represents the interactions between the machine (software to be developed) and its application environment (see Fig. 1). It shows where in the environment the software development problem is located.



a: {phraseTextMessage, sendTextMessage, login, signUpChatSession, signOffChatSession}
b: {showTM, showTMMeta, showTMDefault, showUserID, showCourseOfChat, showTransformationInProcess, showListOfParticipants}
c: {registerTM, recordUserID, CourseOfChatContent}
d: {editTextmessage,TMContent, TMMeta, TMDefault}
e: {followCourseOfChat}
f: {IdentifiesTMOriginator}
g: {Correspond2OneAnother}
h: {relateUser2ID, UserID}
i: {UserList, registerUserID}
j: {AvailableSessions}
k: {collectOfferedChatSessions}
m: {enterUsersIntoListOfParticipants}

Fig. 1. Context diagram of the chat application

A context diagram consists of *domains* (rectangles) and sets of *shared phenomena* (labeled links between rectangles), which are derived from the requirements and domain knowledge (cf. Tab. 1). Domains correspond to entities or facets of the real world, whereas the *machine domain* (rectangle with two vertical lines) represents the software product which ought to be developed, in our case: the chat application itself. Hence, there is exactly one machine domain contained in a context diagram (chat application in the center of Fig. 1). Any arbitrary number of additional domains can be part of the overall problem situation. They can be further classified. Data types, data structures, database schemata, or other representations of information, that need to be built and introduced by the software developer, are denoted by *designed domains* (rectangle with only one vertical line), for instance text-message. *Given domains* (simple rectangles) are concepts of the real world, which already exist and do not need to be constructed. They are relevant for the problem description and its solution, and need to be considered in the context diagram, too, for instance display.

Shared phenomena are operations, actions, events, or states which are common to two domains. For instance, the machine domain chat application shares the phenomenon AvailableSessions (which is derived from requirement R8) with the designed domain sessions, cf. interface j in Fig. 1. Context diagram, requirements, and domain knowledge are created and collected iteratively to cover the overall problem situation and help to understand the given and desired interactions of environment and machine. Requirements and domain knowledge that are found through software analysis are expressed by domains and shared phenomena in the context diagram (cf. Tab. 1 and Fig. 1).

2.2 Decompose Overall Problem into Simple Subproblems

As the context diagram in Fig. 1 illustrates, it would become difficult to start a structured software development process based on such a complex problem situation. The problem needs to be split into simple subproblems for which known solution methods are available. This is achieved by decomposing the context diagram with the help of the requirements by means of *knowledge-based projection* into smaller subproblems. For each subproblem, all other subproblems are assumed as already solved (*separation of concerns*). The subproblems are derived from the context diagram using operators for problem decomposition (e.g., by combining domains or omitting shared phenomena). Those simple and independent subproblems are represented by instantiated problem frames in the following.

3 Patterns for Software Development Problems

Problem frames [9] are patterns to structure and classify software development problems. Each problem frame is represented by a frame diagram, which relates a set of requirements via several problem domains to the machine domain, using shared phenomena. The outcome is a fixed and abstract problem structure.

A problem frame needs to be instantiated by concrete problem content, taken out of the context diagram with the help of requirements. An instance of a problem frame shows the relation of the respective requirements to the particular domains of the corresponding problem context, which are relevant to reflect the requirements.

Requirements describe desired properties of the environment after the machine is in action. In contrast, shared phenomena at the interface of machine and environment are used to formulate the *specifications*, which are descriptions that are sufficient to develop the machine.

Problem frames support the creation of adequate software specifications (e.g., represented using UML sequence diagrams) by elaborating the essential interactions of environment and machine. The specification constitutes the basis for the development of the machine. It is used for software design, coding, testing, and acceptance of the final software product.

Sometimes it is necessary to compose and create new problem frames to be able to detail and classify a certain software development problem more precisely. Therefore, we merged and extended the basic problem frames introduced by Jackson where applicable. Some selected frame instances of the chat application example are presented to exemplify our pattern-based software development approach. In the following, we show the abstract frame diagram elements (in italic style) together with the concrete problem pattern instance (content of a domain and corresponding shared phenomena).

3.1 Fit Subproblems to Problem Frames

Figure 2 shows the instance of the problem frame "*commanded model display*". Its frame diagram is a variant of the "*commanded display*" frame developed by Jackson [9] and an enhancement of the "*query*" frame developed by Choppy and Heisel [3]. The

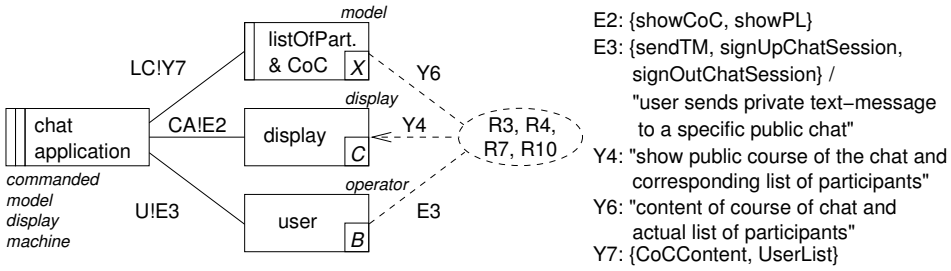


Fig. 2. Instance of the problem frame ”commanded model display”

problem frame ”*commanded model display*” can be composed out of the basic problem frames ”*commanded behaviour*” and ”*model display*”, too.

The problem frame ”*commanded model display*” in Fig. 2 describes the following problem situation: If the *operator* (user) sends a command (phenomenon signUpChatSession of interface E3) to the machine (chat application), it will be executed by the machine and yields some according effects. Here, the state of the *model* (list of participants) is shown on a *display* (display) using the phenomenon showPL of the interface E2. In addition, the actual state of the *model* (public course of the chat) is shown, Fig. 2.

To extract this subproblem from the overall problem, we applied two decomposition operators. We *combine domains*, e.g., list of participants (listOfPart.) and course of the chat (CoC). Additionally, we *omit shared phenomena*, e.g., registerUserID at the interface Y7, because it is not relevant for this subproblem.

Compared to the context diagram in Fig. 1, this frame instance contains only those domains and shared phenomena, which are relevant to fulfill a subset of all requirements namely R3, R4, R7, and R10 (see Tab. 1). The oval on the right-hand side of the frame diagram contains the requirements which are mapped to according parts of the problem context (dashed lines to the domains). The left-hand side of the frame diagram relates the corresponding problem context to the machine. Thus, an instantiated problem frame that is read from right to left indicates the translation of natural-language requirements (in the oval) via the problem context (domains) into technical descriptions which are sufficient to build the machine. Phenomena at the interface of environment and machine can be used to derive *specifications*. The arrowhead pointing to a domain constrains the domain’s behavior or characteristics as stated in the requirements. For example, the requirements R3 and R7 in our application example describe a restriction on the domain display.

Each domain in a frame diagram is marked by a character such as X, C, or B to distinguish the different domain types. The designed domain text message is a *lexical domain* (marked X) which has symbolic phenomena associated to it. The display is a *causal domain* (marked C) which does not need to be built but can be controlled using phenomena, too. The user is represented by a *biddable domain* (marked B). His or her behavior cannot be predicted or controlled by the machine. Indeed, the user can make inputs to the software, but cannot be forced by requirements to act in a predetermined way. However, assumptions as part of the domain knowledge are used to make explicit the expected user behavior (cf. Tab. 1).

Symbolic values are indicated by Y, *events* are annotated with E, and C indicates *causal phenomena*. The characters are numbered for indexing the shared phenomena. The exclamation mark (!) specifies which domain *controls* a shared phenomenon. However, this does not imply control flow. For example, LC!Y7 in Fig. 2 in fact expresses that the merged domain list of participants & course of the chat (abbreviated LC) is responsible for administrating the symbolic phenomena in the set Y7. However, the chat application determines when to query the required information. All subproblems contain exactly one machine domain (cf. Fig. 2, Fig. 3, Fig. 4, and Fig. 6). In contrast to Jackson who gives different names to each machine domain, we prefer identical names (here: chat application) to indicate that they constitute parts of a common machine.

3.2 Identify Quality Characteristics

The problem frames approach and more common software development notations such as UML share the same deficiency: they do not offer adequate notations to record software quality characteristic. Although it is commonly accepted that software quality is mainly reflected by non-functional properties (soft goals) [12], only a few approaches exist to elicitate and document them systematically [4]. Software quality characteristics are difficult to grasp, and its hard to maintain them during the software life cycle appropriately. One reason can be that software quality actually is seen as a global attribute of the overall software product, which cannot be related explicitly to local functionality of parts of the software. The idea that software quality is related to the system as a whole rather than to individual system features can be found likewise in requirements engineering [15] and in architectural design [14]. In our approach, we identify and assign quality characteristics to a local set of software functionality. To do so we use problem frames to represent the local behavior and annotate relevant local quality characteristics to them. We extend Jackson's problem frames approach by explicitly annotating software quality characteristics in frame diagrams.

3.3 Classify Subproblems According to Quality Demands

Based on our chat application example, we show how the basic behavior of a system can be expressed using problem frames and how quality characteristics such as usability and security can be considered by detailing the core software features with the help of special usability and security problem frames.

Usability Engineering using Problem Frames Usability Engineering contributes to the improvement of human-computer interaction (HCI). It takes psychological aspects into consideration to support the design of software and user interfaces that are easy to use. Although various usability techniques (guidelines, standards, and patterns) exist, they lack of systematical applicability, because often no technical description is available. Some authors of HCI design patterns [16, 13] refuse a technical detailing of patterns to keep them comprehensible for non-experts. In contrast, other authors [5] transformed HCI design patterns into UML, but do not offer a process of how and when to apply them exactly. There is an urgent need to integrate usability aspects into

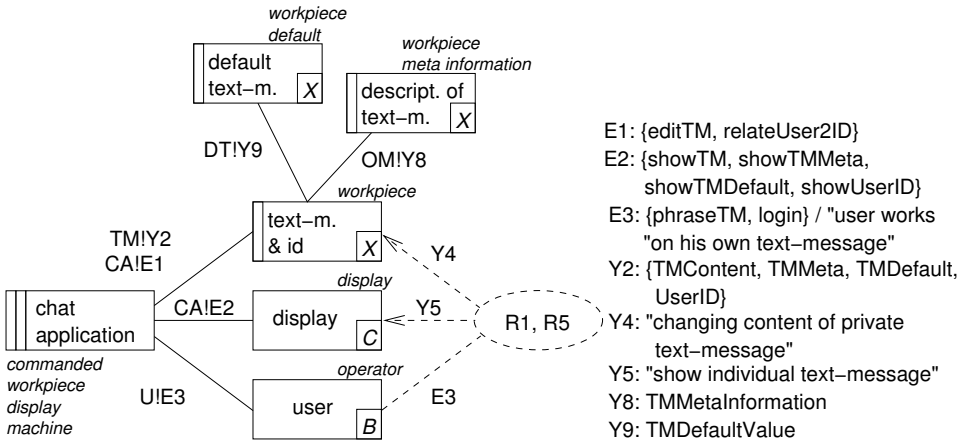


Fig. 3. Instance of the *HCI*Frame ”commanded workpiece display”

the software development process. To bridge the gap between informal descriptions of HCI design patterns and the wish to apply usability concerns systematically during the software development process, *the problem descriptions* of HCI design patterns are used in requirements analysis using HCI-oriented problem frames (*HCI*Frames)[18]. *HCI*Frames allow to identify and express usability demands already in the early software development phases. As we will show, usability problems which can be elicited and documented in software analysis can be considered in software design more easily and finally lead to a software product which realizes software quality requirements in a traceable way.

Instantiating HCIFrames Figures 3 and 4 show selected subproblems that describe different aspects of the given problem situation. They already consider usability requirements. The problem frame ”commanded workpiece display” in Fig. 3 which is a composite of the basic problem frames ”simple workpiece” and ”commanded behaviour” from Jackson is extended by HCI-related problem descriptions taken from the HCI design patterns of Tidwell [16], namely *input hints*: ”place a sentence to explain what is required” (*workpiece meta information*) and *input prompt*: ”prefills telling the user what to do” (*workpiece default*). The two new domains default text-message (indicating that a initial text-message should have a default value like ”Hello World!”) and description of text-message (requiring a label or explanation of what kind of input is expected from the user, for instance ”type your chat message here:”) which are related to the *workpiece* text-message. A *workpiece* is a lexical domain that can be altered.

In fact, a software development problem that fits into a problem frame containing a *workpiece* can explicitly describe quality characteristics like in this example for usability, if the *workpiece* domain is extended by *workpiece meta information* and a *workpiece default* which support self-explanatory user interfaces.

The problem frame ”commanded transformation” in Fig. 4 consists of the basic frames ”commanded behaviour” and ”transformation”. It is extended by the problem

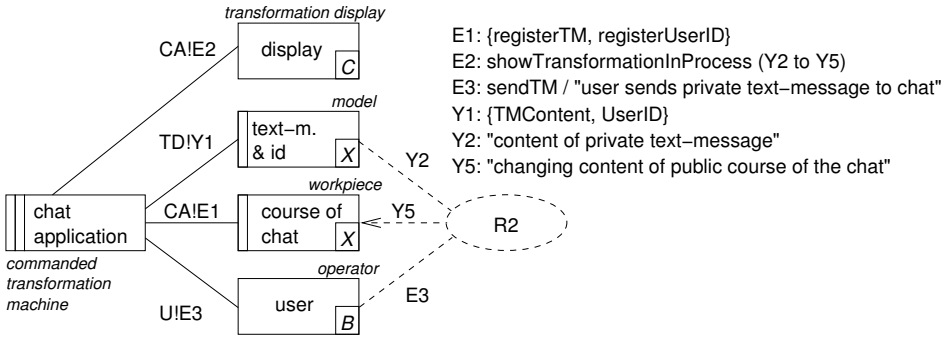


Fig. 4. Instance of the *HCI*Frame ”commanded transformation”

domain *transformation display*, which reflects the *problem description* of the HCI design pattern *progress*: ”tell user whether or not an operation is still performed and how long it will take” from van Welie [10]. The meaning of this *HCI*Frame is that whenever there is a transformation in progress (machine internal working process represented by a transformation problem frame) this is indicated to the user. For instance, in the following software design we decide to realize this transformation information by a progress bar or an information like ”sending your text-message” or ”please wait...operation in process” on a *transformation display* (display). In software analysis, we are only interested in identifying this usability requirement, whereas in software design, we will decide on its implementation. Figures 3 and 4 show that software quality aspects such as usability can be expressed with the help of *HCI*Frames. After we have introduced problem frames to express security requirements, we show how the final instantiated problem frames for our chat application example can be mapped to patterns of software architecture and design, preserving all specified quality characteristics.

Security Engineering using Problem Frames To meet the special demands of software development problems occurring in the area of security engineering, we are developing a catalog of security problem frames considering different security problems [8], [7]. Security problem frames consider *security requirements*. The goal is constructing a machine that fulfills the security requirements. The security problem frames strictly refer to the *problems* concerning security. They do not anticipate a solution. For example, we may require the confidential transmission of data without being obliged to mention encryption, which is a means to achieve confidentiality. *Solving* a security problem is achieved by choosing generic security mechanisms (e.g., encryption to keep data confidential). For this purpose we are developing a catalog of concretized security problem frames [8], [7]. They consider *concretized security requirements*, which take the functional aspects of a security problem into account. For each of the developed security problem frames there is at least one concretized counterpart providing a generic security mechanism. The security problem frame and its concretized counterpart used in this paper serve to treat the security requirement of *anonymity* and the concretized security requirement of *pseudonymity*, respectively.

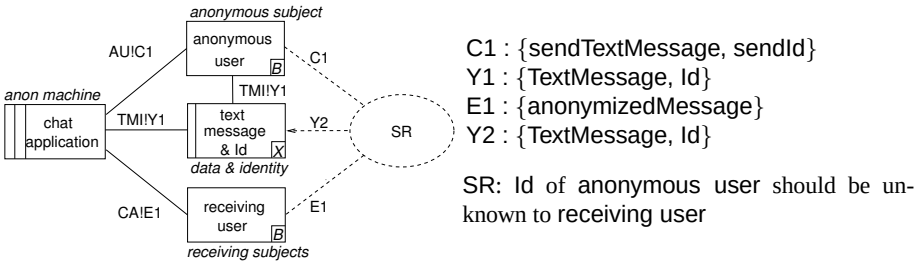


Fig. 5. GUI anonymity frame diagram

Instantiating Security Problem Frames We now extend the requirements to be considered when developing the chat system by the security requirement *anonymity of the chat participants*. Anonymity of users and other systems is an important issue in many security-critical systems. Anonymity is the state of being not identifiable within a set of subjects [11].

Anonymity can be considered from different views, e.g., anonymity on the level of the graphical user interface (GUI), and anonymity on the level of the network. In this paper, we focus on GUI anonymity. Figure 5 depicts the security problem diagram for GUI anonymity. It is an instance of the underlying security problem frame for anonymity, which is not depicted separately in this paper.

The problem diagram in Fig. 5 contains the machine domain chat application. The lexical domain text message & Id represents the sent text message including the real id of the sender represented by the biddable domain anonymous user. The text message is received by the other chat participants represented by the biddable domain receiving user. Both, anonymous user and receiving user are specializations of the domain user (see Fig. 1). The security requirement SR states that the receiver of the text message should not know sender's real Id.

Resolve conflicting Quality Characteristics The chat application's functional requirements and quality characteristics are now identified and described. In order to be able to derive a specification, we must ensure that the elicited requirements do not contain any conflicts. When checking for conflicts it becomes apparent that the requirement R5 (cf. 1) "Each text-message is related to its respective user so that the originator of a message can be identified." is at odds with the required SR (cf. 5) "anonymity of the chat participants". A convincing compromise to resolve this conflict can be found by negotiating both requirements. As a result, we decide to choose a generic security mechanism that uses pseudonyms [11]. With this approach, the text messages can be related to their originators (represented by pseudonyms), and at the same time the originators' identity is kept confidential.

Instantiating Concretized Security Problem Frames In the course of transforming the security requirement for GUI anonymity into a concretized security requirement, the domain text message & nick name (see Fig. 6) is introduced.

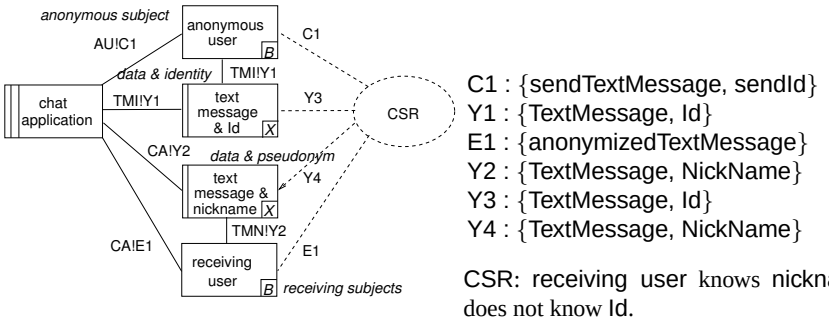


Fig. 6. Concretized GUI anonymity frame diagram

The domain part nickname represents a pseudonym. A pseudonym is a string which, to be meaningful in a certain context, is

- unique as ID
- suitable to be used to authenticate the holder and his/her "items of interest" (e.g., messages and network packets)

The holder of a pseudonym must be linked to the pseudonym itself. Then, the links must be kept confidential in order to achieve anonymity. In Fig. 6 the links are administrated by the machine domain chat application. Therefore, we may assume that chat application will keep the links confidential. The concretized security requirement CSR in Fig. 6 constrains the domain text message & nickname. The domain part nickname should be known to the receiving user, while the domain part Id should be unknown to them.

4 Role-Driven Mapping of Requirements Analysis Documents to Architectural Design Artifacts

Whatever a pattern is used for (in software analysis or design, in security or usability engineering), it generally assigns roles to entities or facets and their interaction in an abstract fashion. Patterns need to be instantiated to specify who take a certain role and to bring them into action for a concrete situation. We make use of this observation to match patterns used in requirements analysis with patterns of software architecture and design.

Starting with the problem frame instance in Fig. 2, we identify the different roles taken by the respective domains and shared phenomena. Three roles can easily identified from the problem frame diagram, namely *operator* (user), *display* (display) and *model* (list of participants & course of the chat). Now, we search for a corresponding architectural style or design pattern reflecting these roles.

Figure 7 shows the architectural style "Model-View-Controller" [2] in its upper half. The classes of this architectural style can be regarded as descriptions of the roles that objects of these classes can take. Accordingly, we map the roles *model*, *display*,

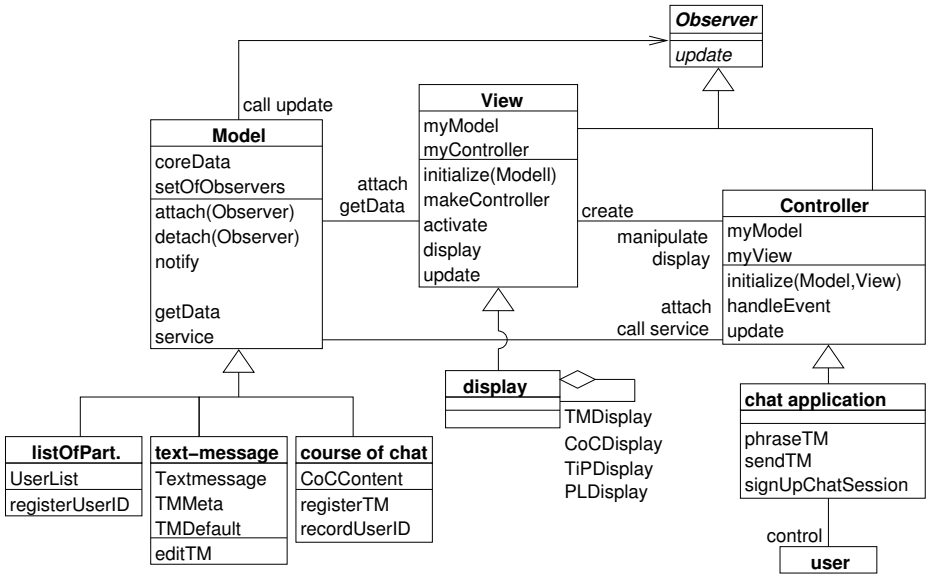


Fig. 7. Instantiated architectural style ”Model-View-Controller” for the chat application

operator of our problem frame to the classes Model, View, Controller of the architectural style. This mapping is appropriate, because the domain roles of the problem frame and the roles of the corresponding classes in the architectural style are comparable. Since the problem frame in Fig. 2 is instantiated, we can reuse its concrete domains and shared phenomena to instantiate the chosen architectural style, too. The lower half of Fig. 7 shows how the domains and shared phenomena are mapped to the respective classes of ”Model-View-Controller” via an inheritance relation, which can be used in the UML to express assignment of roles. The chat application becomes the controller, because the machine operates according to the user commands. For the given problem situation in Fig. 2, we found one possible design which satisfies its requirements R3, R4, R7, and R10 for the chat application example.

To illustrate how quality characteristics can be preserved from requirements engineering to software design we consider the *HCI*Frame instance of ”commanded workpiece display” in Fig. 3 in detail. Similar to the previous problem frame instance of Fig. 2, it can be mapped to model-view-controller, because the role of a *workpiece* is comparable to the role of a *model*, both relying on being processed by the machine. To trace the quality characteristics stated in the *HCI*Frame instance of Fig. 3, it is necessary to consider the class text-message in Fig. 7 in more detail. The default text-message and the description of the text-message were translated from the interfaces Y8 and Y9 of the usability requirements into the interface Y2 containing TMDefault and TMMeta of the specification in Fig. 3. The latter have become attributes of the class text-message in Fig. 7. The frame instance of ”commanded transformation” in Fig. 4 requires an additional View in order to consider the *transformation (in process) display* domain display. The usability requirement which demands a *transformation display* is considered by

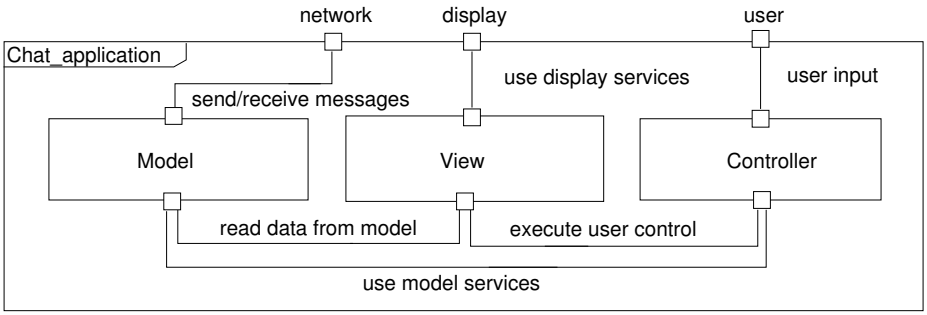


Fig. 8. Global architecture of the chat application

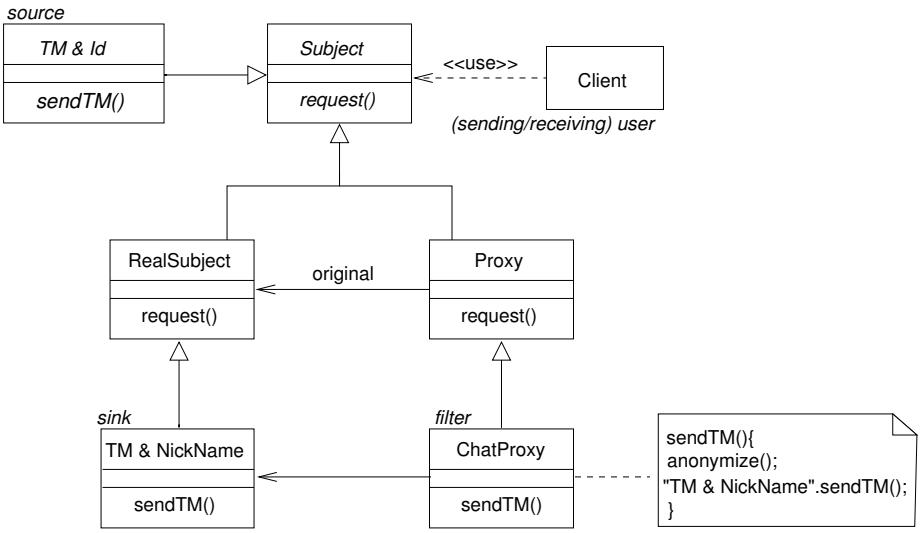


Fig. 9. Proxy design pattern/instance for GUI anonymity

the rolename TiPDisplay at the class display of Fig. 7. The role-driven mapping of the *HCI*Frame instances in Fig. 3 and Fig. 4 shows that all specified quality characteristics are preserved.

The architectural style ”Model-View-Controller” is instantiated to become the global architecture of the chat application. This architecture is depicted in Fig. 8. The ”Model-View-Controller” architecture consists of the three components Model, View, and Controller, which themself have an architecture.

Analyzing the security requirements concerning the GUI anonymity in Sect. 3.3 shows that the machine to be developed must be able to act like a *placeholder*. The text messages including the Id of the user are received by the placeholder. Then, the placeholder is responsible for exchanging the user’s Id by a pseudonym and sending the text messages and the pseudonym to the receiving chat participants. Because the behavior described above can be generally observed when applying the concretized

security problem frame for anonymity using pseudonyms, we link this frame to the design pattern *Proxy* [6] in combination with the architectural style *Pipe-and-Filter* [1].

The ”Proxy” design pattern (in combination with its instance for GUI anonymity) is depicted in Fig. 9. The ”Proxy” design pattern is a structural pattern that introduces a placeholder (Proxy) in order to control access to the originator *Subject*. Hence, we can map the domains of the concretized security problem diagram shown in Fig. 6 to the components of the ”Proxy” pattern. The domain text message & Id is represented by the class TM & Id, the domain text message & nickname is represented by the class TM & NickName, and the machine domain chat application is represented by the class ChatProxy. The domains Anonymous user and Receiving user are represented by Client. ChatProxy receives the text messages including the chat participants's Id. Then, the ChatProxy anonymizes the received data and forwards the text message including a nickname to the actual receiving chat participants.

When anonymizing received data, the ”Pipe-and-Filter” architectural style comes into play. It sees a system as a series of filters (or transformations) on input data. Data enter the system and then flow through the components one at a time until they reach some final destination. Filters are connected by pipes that transfer data. We consider the linear pipeline, in which each filter has precisely one input pipe (*source* in Fig. 9) and one output pipe (*sink* in Fig. 9). Additionally, only one filter (*filter* in Fig. 9) is needed to exchange an Id by a nickname. This functionality is reflected by the function anonymize().

Both, the ”Proxy” and the ”Pipe-and-Filter” architectures describe the internal architecture of the component View in Fig. 8.

Role-driven mapping enables a smooth transition of patterns used to represent the problem in software analysis to patterns used to represent a solution detailed by architectural software design. In particular, role-driven mapping serves to preserve quality characteristics.

5 Conclusion

We have shown that functional requirements as well as quality characteristics such as security and usability can be treated using our extension of Jackson's problem frames approach. We presented a software development method that preserves usability and security quality characteristics using a role-driven mapping of requirements analysis documents to architectural design artifacts.

With this approach, software engineers can hope to cover large parts of the early phases in software development using patterns.

In the future, we intend to find new patterns to extend the catalogs of *HCI*Frames, security problem frames, and concretized security problem frames. Furthermore, we intend to apply our approach to other quality characteristics such as performance and scalability.

Additionally, we plan to elaborate more on the later phases of software development. For example, we want to investigate how to integrate component technology in the development process. Finally, we plan to provide tool support for our pattern-based software development method.

6 Acknowledgements

The authors appreciate the in-depth comments given by the anonymous reviewers to improve this work. We would like to thank our doctoral thesis supervisor Prof. Dr. Maritta Heisel for her support.

References

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

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

[3] C. Choppy and M. Heisel. Une approache à base de patrons pour la spécification et le développement de systèmes d'information. *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL*, 2004.

[4] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulus. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, Boston, USA, 2000.

[5] Eelke Folmer and Martijn van Welie and J. Bosch. Bridging Patterns: An approach to bridge gaps between SE and HCI. *Information and Software Technology*, 48(2):69–98, 2006.

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

[7] D. Hatebur, M. Heisel, and H. Schmidt. Pattern- and Component-Driven Security Engineering. Technical report, Universität Duisburg-Essen, 2006. `http://swe.uni-duisburg-essen.de/intern/seceng06.pdf`.

[8] D. Hatebur, M. Heisel, and H. Schmidt. Security Engineering using Problem Frames. In G. Müller, editor, *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, LNCS 3995, pages 238–253. Springer-Verlag, 2006.

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

[10] M. van Welie. Patterns in Interaction Design, 2003-2006. `http://www.welie.com` Online catalogue for interaction design patterns.

[11] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In H. Federrath, editor, *Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2001 / 2009, pages 1–9. Springer-Verlag, 2000.

[12] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, Boston, USA, 1999.

[13] T. Schümmer. *A Pattern Approach for End-User Centered Groupware Development*. PhD thesis, FernUniversität Hagen, 2005.

[14] M. Shaw and S. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, Eaglewood Cliffs, New Jersey, USA, 1996.

[15] I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[16] J. Tidwell. *Designing Interfaces*. O'Reilly Media, Sebastopol, USA, 2005.

[17] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, August 2005. `http://www.uml.org`.

[18] I. Wentzlaff and M. Specker. Pattern-Based Development of User-Friendly Web Applications. In *Proceedings of the 2nd International Workshop on Model-Driven Web Engineering (MDWE 2006)*, Palo Alto, USA, 2006. ACM.