Chapter 5
Tacit Knowledge in Software Evolution

Jan Ole Johanssen, Fabien Patrick Viertel, Bernd Bruegge, and Kurt Schneider

Requirement elicitation is an essential activity to identify functional and non-functional requirements of a software system. In long-living software systems, requirements identification and update are particularly challenging. This typically results in an incomplete set of requirements. The reasons for this lie in continuous changes over the lifetime of the software system, followed by a substantial part of the requirements that remains unspoken: Users, and generally any stakeholder of a software system, might not be consciously aware of new or evolved needs or of the associated reasons. As a result, they are unable to express and verbalise requirements that relate to this knowledge, which is called *tacit knowledge*. This chapter details the identification and externalisation of tacit knowledge during both the design time and run time of a long-living and continuously evolving system. The overall goal is to detect deviations between explicitly elicited requirements and implicitly derived requirements. We discuss two cases in which the identification and externalisation of tacit knowledge is crucial for high-quality software systems.

In the first case, tacit knowledge about security is identified and externalised by heuristics as an example for non-functional requirements elicited during design time. Previously externalised knowledge is encoded in heuristics and filters for machine learning, which classify general requirements into more and less security-related ones. As a consequence, security experts can focus their time and effort on the more security-related requirements. In the long term of a long-living software system, externalising and reusing tacit security knowledge will be embedded in a cyclic learning process.

J. O. Johanssen (✉) · B. Bruegge
Technische Universität München, Institut für Informatik I1, Garching, Germany
e-mail: jan.johanssen@tum.de; bruegge@in.tum.de

F. P. Viertel · K. Schneider
Leibniz Universität Hannover, Fachgebiet Software Engineering, Hannover, Germany
e-mail: fabien.viertel@inf.uni-hannover.de; kurt.schneider@inf.uni-hannover.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_5

The second case focuses on tacit knowledge captured during the run time of a system to improve the functional aspects of a software system. Usage monitoring allows to understand the difference between the specified and observed behaviour of a user. A system is inconsistent or incomplete if the requirements are incorrectly implemented or an important feature has not yet been identified and implemented. Traditional approaches address these problems only by using bug reports and change requests. We claim that the identification and extraction of tacit usage knowledge help to reveal misunderstandings and leads to feature requests without the active verbalisation by the users of the software system.

5.1 Toward Identification and Extraction of Tacit Knowledge

Software systems are built on a set of requirements established during requirements engineering. Requirements elicitation is a major activity of requirements engineering aiming at a complete representation of the system under development and its *external* behaviour [Dav93].

Long-living systems face challenges even if state-of-the-art requirement elicitation practices are applied. A component without confidential data but with Internet access may turn into a security-related one when it is connected to yet another component that contains customer data. Likewise, a simple view may be easy to use in its initial version, but during the system's lifetime, new visual components are added, affecting the way the interface was originally designed. Each set of requirements may look simple by itself; however, in combination, they may require specific attention. Even a system that is initially considered secure or user-friendly may eventually become vulnerable or confusing by the continuous changes of the long-living system.

Developers may have some understanding of security or usability concerns but only a very limited knowledge for recognising related aspects. When they implement new functionality or integrate components, they may not recognise implicit vulnerabilities or usability problems. They would need a hint or *breakdown* to raise their attention. In addition, existing requirements relate to and have an impact on more aspects than initially defined. At the same time, the attempt to obtain a complete specification of requirements often leads to analysis paralysis [Bro+98]: The intention to analyse an aspect in its entirety slows down the process and finally paralyses it. In this chapter, we focus on the aspects of the following requirements:

- **Non-functional requirements** and their impact that neither customers nor developers are aware of during requirements elicitation at design time.
- **Functional requirements** that evolve during the run time of a software system that end users are unable to express.

Tacit knowledge is knowledge deeply ingrained in a person's mind [PS09]; a person will apply such knowledge repeatedly but may not be able to verbalise this given knowledge. For example, security experts avoid code injection vulnerabilities

as part of their expertise. Likewise, developers keep the user interface simple and easy to use without explicit requirement. When their activity of competence is interrupted while they apply this tacit knowledge—the breakdown—they will remember the rationale. In many cases, domain experts are not even aware that their expertise depends on this knowledge and that this knowledge might be useful for others. We follow the hypothesis that the utilisation of tacit knowledge allows the requirements of long-living systems to be kept consistent and complete throughout the lifecycle of the system. We analyse two perspectives on tacit knowledge: design time and run time.

A Design Time Perspective on Requirements. Systems evolve over time. During the initial design phase, certain aspects might be considered irrelevant. For instance, a supermarket system designed without the Internet in mind would not consider attacks or vulnerabilities that arise when the system is extended to an online store during its evolution. Thus, security requirements and the awareness for security-related aspects of functional requirements may have not been considered during the initial design phase. To cope with this situation, developers extend the functionality but often overlook the need to adapt associated non-functional requirements, such as security, that result from the change. Over time, this neglect will turn an initially secure system into an insecure and vulnerable one.

A Runtime Perspective on Requirements. Information on users and on how they practically employ a system might not be present during requirement elicitation. Therefore, systems might not deal well with users, and previously made decisions require refinements. In addition, new requirements are demanded since they become relevant only when the software is used during a later point in time. Users and their intention change over time, which results in changed requirements that evolved by frequently using the software. Two approaches were developed to handle the lack of usage knowledge, that is how software is being utilised by end users: To support requirements elicitation, the concept of a stakeholder was introduced to software engineering [Con94]. Stakeholders represent the interests of clients, customers, and developers—but often neglect the interests of end users and are difficult to identify if a user has not been able to participate in the requirements elicitation [SFG99, Con94]. In the field of human-computer interaction—and in other fields such as marketing [Jen94]—another approach was established to deal with not yet existing users: personas, so-called "hypothetical archetypes" [Coo99], refer to a fictional and synthetic character that one would imagine a user *could* look like, focusing on certain characteristics. Personas are derived from a limited population sample and reflect specific characteristics of users.

We present evolutionary approaches for both perspectives, namely to identify neglected non-functional requirements, such as security during design time, and to identify functional requirements by observing real users during run time. Both approaches share similar challenges: discover, understand, and transform users' tacit knowledge into explicit knowledge.

To transform tacit knowledge to explicit non-functional requirements during design time, we describe an approach that identifies security-related requirements semi-automatically using natural language processing. Our approach is able to

retrieve vulnerabilities from requirements written in natural language based on security incidents.

We describe a formative approach for understanding users from runtime information, which begins with personas as the starting point for the classification of real users. This is similar to a *greedy* algorithm, which starts with a local optimum—an assumption of how a hypothetical stakeholder [RC03] could look like—and continues searching for a better user understanding.

Both approaches apply iterative and evolutionary procedures. We begin with an empty starting situation, for example knowing nothing about security requirements or the user's preferences. Both approaches aim to improve the current set of requirements for a given problem and the understanding of the real users by continuously transforming tacit knowledge into explicit knowledge. Ultimately, this process will result in:

- Increasing the system's usability and customisation towards the needs of users by software releases that better fit the requirements of customers and the expectations of users
- Improving and maintaining the quality of development for long-living systems by co-evolving non-functional requirements, such as security or usability

The chapter is structured as follows. In Sect. 5.2, we provide an overview of the foundations of tacit knowledge. In Sects. 5.3 and 5.4, we introduce our approaches and highlight their application in a concrete example. The approaches address the two main challenges as described in Sect. 3.1: identification and extraction of tacit knowledge, as well as detection of deviations in requirements. In Sect. 5.5, we present related work. Section 5.6 provides a summary, outlook, and suggestions for further reading.

5.2 Foundations

The aim of software engineering is to establish activities for specifying, developing, and managing software evolution. However, these activities usually cannot capture every aspect required for a complete specification. One reason for the incompleteness of the specification lies in the inability of stakeholders to express their requirements—even though they are aware of a *need*, generally referred to as *tacit knowledge*.

Polanyi builds his definition of tacit knowledge on the fact that "we can know more than we can tell" [PS09]: In his book *The Tacit Dimension*, he further coins the term tacit by describing it as a skill, positioning the term closely to physical actions such as riding a bicycle or playing an instrument—actions that are learned over a long period and apparently impossible to describe in words. Polanyi systematically describes the inner workings of a human when experiencing or, more precisely, externalising tacit knowledge. He identifies the *functional* relationship and structure of tacit knowledge, which allow to disassemble the individual parts of

tacit knowledge. Further, semantic and ontological aspects lead to the *phenomenal* structure of tacit knowing.

Gigerenzer [Gig08] uses the comparison of a native speaker that—while they can find a sentence to be grammatically correct—they are usually unable to verbalise the underlying grammar; he calls this *gut feeling* and uses the term interchangeably with intuition and hunch [Gig08]. Gigerenzer continues to exemplify that humans tend to choose logically unlikely alternatives when asked for predicting the likelihood of two alternatives—the *conjunction fallacy*. They base their decision on impressions rather than mathematical *rationale* [Gig08].

In his book *The Reflective Practitioner: How Professionals Think in Action*, Donald A. Schön recognises similar patterns in working environment settings [Sch83]. He coins the phrase that *our knowing is in our action* [Sch83]. He develops the term *tacit knowing in action* by noticing that practitioners are continuously making decisions during their day-to-day work, such as the assessment of situations or quality criteria, without paying attention to the act of decision-making. However, sometimes they are interrupted during this process and reflect on their action: By extracting the underlying features of their judgements to criticising existing approaches, they arrive at an improved embodiment [Sch83].

Nonaka and Takeuchi provide an extensive examination of the differences between *explicit* knowledge, that is written down in rules, definitions, or handbooks, and *implicit* knowledge, that is experiences of an individual that are based on personal values and motivated by cultural aspects [NT12]. In their book *The Knowledge-Creating Company*, the authors describe the dynamic interplay between these two knowledge types as the key for knowledge creation in companies. They establish a spiral model that contributes to the social process of knowledge sharing that heavily depends on a collaborative interaction and leads to the externalisation of knowledge, which makes it useful for companies.

Tacit knowledge is investigated in multiple fields, such as social, psychological, or physiological science. Understanding and externalising tacit knowledge can be valuable for other disciplines as well. For instance, Schneider acknowledges that specific techniques are needed to capture requirements and additional information when and where they surface: in natural language requirements specifications or by observing activities by experts [Sch09].

5.3 Tacit Knowledge During Design Time

Tacit knowledge is not easily available for extraction, externalisation, and use by others. A person with tacit knowledge acts in a knowledgeable way but is not able to explain that knowledge. In the first part of this section, we describe a case in which requirement engineers and developers deal with requirements. Since they are usually not security experts, their experience in security is limited. Security experts are knowledgeable about security but may be unable to apply that knowledge to a given set of requirements. A large part of their security knowledge remains tacit.

They need a breakdown in order to shift tacit knowledge to their conscience and apply it. In the following heuristic approach, we use natural language processing, ontologies, and frames to guide and focus the attention of security experts to use cases (UCs) that are more security-related than others. This is supposed to reduce their effort and help them focus on the most rewarding requirements for identifying security problems.

This focus and contextualisation can help to externalise their respective tacit knowledge. The externalised knowledge will also be stored for future use: It can help improve the above-mentioned heuristic filtering mechanisms, thus improving the automated part of classification.

5.3.1 Security in Requirement Documents

Security is an important quality aspect. It is not obvious whether a requirement is security relevant or not. It will depend on other requirements and on the environment that the software is used in: Depending on laws, different levels of security will be required. Knowledge about security incidents or innovations in attacks has a major influence on security. All these aspects are in constant flux and need to be monitored to keep a long-living system secure.

Most customers and requirement engineers are not security experts. In the requirement elicitation phase, some of them rely on their *gut feeling* in judging the security relevance of requirements. This gut feeling or *experience* indicates certain knowledge that is, however, difficult to grasp. Developers consider a requirement security-related, but they cannot say why. It just looks suspicious to them. From their perspective, the reason for that suspicion is *tacit knowledge*.

Use cases and a specification document are artefacts resulting from requirement activities. Use cases support the understanding of requirements and describe what the system should do. In most cases, they are written in natural language, which makes them more comprehensible for customers. Due to the large number of requirements and use cases involved in a large long-living software system, checking entire specifications and all use cases for security concerns would be very laborious and, in most of the cases, impossible for economic reasons.

Therefore, we developed a semi-automatic approach for the classification of natural language requirements with a special focus on use cases. As a result, only parts of those artefacts are classified as security-related and then need an in-depth investigation by security experts.

Even security experts cannot cover all relevant security knowledge to determine whether a requirement is security-related or not. While developers and requirements engineers are not aware of security concerns, security experts may not be able to identify a concrete problem with respect to their large internalised knowledge about potential attacks. Again, a lot of tacit knowledge needs a breakdown to come to the foreground.

This observation led us to the following research questions [Gär+14]:

- **RQ1**: How can security knowledge be organized in a way that it can be used for assessing the requirements of a long-living software system?
- **RQ2**: How can requirements engineers identify security-critical issues in natural language requirements semi-automatically?
- **RQ3**: How can requirements engineers be supported to extract proper security knowledge from identified security-critical issues in requirements?

We need a security knowledge model to use and collect security-related knowledge. Our approach uses heuristics to identify security vulnerabilities.

Our goal is to support the security assessment of requirement while using security knowledge of reported security incidents. We focus on use cases. In Sect. 5.3.2, we show how related knowledge is modelled. Requirements are classified semi-automatically. Among other techniques, we use Natural Language Processing (NLP). The classification is performed based on the semantic of words in a requirement. In Sect. 5.3.3, the approach is described in detail. We describe the identification of security issues by heuristics in the remainder of this section. Furthermore, we explain the extraction of security knowledge from informal sources, such as conversations. The knowledge base is filled from those sources. Section 5.3.4 presents an evaluation on the case study using the iTrust medical health care system.

5.3.2 Modelling of Security Knowledge

Security faces the challenge of unknown unknowns [MH05]: *we do not even know what we don't know*. It is impossible to say which knowledge will be relevant in the future. Relevant security knowledge, for example on new attacks, changes rapidly over time.

Trustworthy data should be securely encrypted. Data Encryption Standard (DES) met this requirement. In the mid-nineties, attacker knowledge revised that perception. Nowadays, DES is considered insecure, so that another encryption such as the extension Advanced Encryption Standard (AES) must be used to meet the above-mentioned requirement of *securely encrypting* data. To prevent a leak of data integrity, we use reasoning techniques to detect these data flows. A detailed description of this procedure is provided in Sect. 5.3.3 Therefore, security knowledge must be maintained by human interaction iteratively.

Security Ontology

Security knowledge consists of knowledge about security incidents, operator obligations, and security guidelines—to name just a few. We collected various taxonomies and ontologies for modelling incident-centric security knowledge from literature

and derived an ontology covering the most important parts. According to Schreiber, there are general ontologies and domain- and task-specific ontologies [Sch08]. The creation of ontology includes the definition and hierarchical ordering of important terms, their properties and relations, as well as their instances.

Our ontology is derived from literature and is a general security ontology. The upper part of that ontology consists of generic terms and concepts related to security, such as assets, vulnerabilities, and attacks. The lower part of the ontology details those concepts with respect to the specifics of a given long-learning system. For example, customer data are considered an asset, and the WiFi connection in a CoCoME store may cause vulnerability.

For identifying the hierarchical structure of the upper ontology, a systematic literature review was applied to identify security-related terms and their relations. We addressed publications about concrete ontologies of security knowledge from the area of threat modelling, risk analysis, computer and network security, software vulnerabilities, and information security management. Furthermore, we consider publications covering information systems, cyber-physical systems, distributed systems, and agent-based systems. The named security concepts of these publications are considered for the concepts of our security ontology. To focus on security issues in requirements engineering, publications should primarily consider the technical security aspects of systems (e.g. protocols and encryption algorithms). Further publications that describe applicable approaches were considered for capture and enrich security knowledge. For the automatic search on digital libraries we used the terms security, information system, software, ontology, and meta-model. To find similar work that we did not find within the automatic search, the references of the found work was checked for relevance. Publications until the beginning of February 2015 were considered. All found publications were selected based on the criteria in the following steps.

First step

- Publication exists in full text and is written in English.
- Publication describes a realised, practical applicable approach.
- Publication addresses the modulation, application, or acquisition of security knowledge in software engineering.

Second step

- Publication describes the terms of an ontology with respect to security and their relations.
- The ontology presented in the publication is universally valid.

Third step

- The ontology describes a specific approach to capture knowledge related to security.
- A concrete knowledge source is considered for the extraction of knowledge.

Table 5.1 Publications considered for the creation of ontology [Gär+14]

Publication	Principal security concepts
Howard et al. [HL98]	Action, target, access, tool, vulnerability, result, objective, attacker
Jung et al. [JHS99]	Asset, vulnerability, threat, security control, risk probability, asset value, impact, EC environment
Mouratidis et al. [MGM03]	Constraints, secure entity (goals, tasks, resources), secure dependency
Undercoffer et al. [UJP03]	Attack, system component, input, consequence, means, location
Alvarez et al. [ÁP03]	Entry point, vulnerability, service, action, input length, http headers, http verb, target, scope, privileges
Swiderski et al. [SS04]	Asset, entry point, trust level, attack, attacker, vulnerability, countermeasure
Herzog et al. [HSD07]	Asset, threat, vulnerability, countermeasure
Tsoumas et al. [TG06]	Asset, risk, threat, attack, threat agent, vulnerability, impact, countermeasure, controls, security policy, stakeholder
Karyda et al. [Kar+06]	Asset, countermeasure, objective, person, threat
Barnum et al. [BS07]	Vulnerability, weakness, method of attack, attack consequence, attacker skill, solution and mitigation, resource, context
Fenz et al. [FE09]	Asset, organisation, security attribute, threat, threat source, threat origin, vulnerability, control, severity scale
Elahi et al. [EYZ09]	Vulnerability, effect, attack, security impact, malicious goal, attacker, countermeasure, malicious action, component, actor
Simmons et al. [Sim+09]	Attack vector, operational impact, defence, informational impact, target (network, application, etc.)
Guo et al. [GW09]	Attack, countermeasure, consequence, attacker, vulnerability, IT product
Miede et al. [Mie+10]	Attack, countermeasure, asset, vulnerability, threat, security goal
Eichler [Eic11]	Asset, threat, damage scenario, protection requirements, safeguard, module

The resulting publications, including their security concepts, are listed in Table 5.1. We identified ontology assets, entry points, trust level, system components, attack, vulnerability, threat and countermeasure. These components are mentioned in several of the considered publications, which leads to the structure of our ontology. In the following, the components and their relations will be described and explained with examples.

- An asset is an item of interest worth being protected (e.g. username and password).
- Entry points define the interfaces to interact with the system. They provide access to assets (e.g. login website, email, input field).
- A trust level describes which role has access to an asset using a specific entry point (e.g. user, administrator).

- System components model the regarded system focusing on assets and entry points. This includes hardware, as well as software components (e.g. database, logging).
- An attack is a sequence of malicious actions that are performed by an attacker aiming at assets (e.g. cross-site scripting, denial-of-service attack).
- Vulnerability is a system property that facilitates unintended access or modification of assets. It violates an explicit and implicit security policy. Entry points may have or provide access to vulnerabilities (e.g. improper neutralisation of input, missing encryption of sensitive data).
- A threat is the possibility to perform a successful attack on a specific asset. Successful attacks exploit at least one vulnerability to cause damage (e.g. execute unauthorised code or commands, expose sensitive data).
- A countermeasure mitigates a certain threat by fixing the respective vulnerability (e.g. input validation, encryption of sensitive data).

In Fig. 5.1, the upper parts of the ontology are displayed. This upper security ontology has to be refined in terms of concepts and in terms of instances. For example, there are various assets of a system, such as username or password, that have to be considered. The concept assets of the ontology have to be instantiated by these concrete assets.

Representation of Knowledge

To monitor different knowledge sources, it is necessary that the knowledge they provide is represented in a uniform manner. Each knowledge item, such as security incidents and use cases, has to be transformed into separate analysis models. They form the so-called security abstraction model. A security abstraction model represents a scenario that describes the use case with respect to security. It has been defined based on the knowledge structure of our security ontology. As an example, the description of a use case is that "a user enters his password into the web form". It contains "user" as trust level, "password" as the related asset, and "web form" as

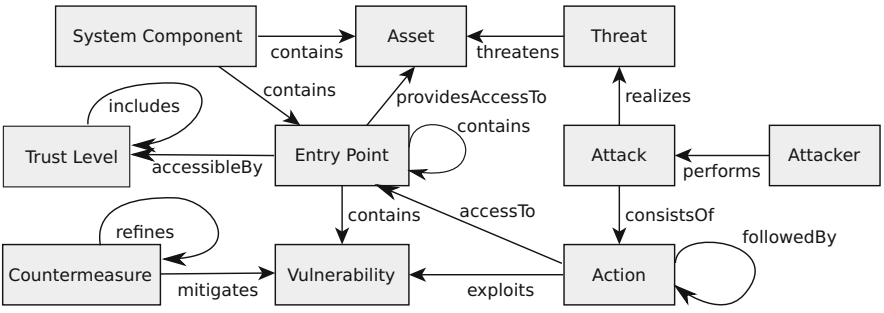

Fig. 5.1 Security ontology [Gär+14]

Table 5.2 Step of use case in an abstraction model

Concepts	Use case
Trust level	User
Assets	Password
Entry point	Web form

the entry point to that asset. Table 5.2 shows the textual representation of a use case step. The model can contain one to multiple of these scenario steps.

5.3.3 Identification and Extraction of Tacit Security Knowledge

In general, our approach consists of two steps: (1) the identification of security vulnerabilities in requirements and (2) the extraction and enrichment of security-related knowledge. For applying our approach, the security ontology has to be manually enriched with security-related terms and their relations to each other by a domain or security expert. To consider requirements and security incidents for the security assessment, the transformation into the previously mentioned analysis model is necessary. These models will be generated automatically in the security assessment approach with the consideration of word similarity and relations between words which are part of both a use case and the ontology.

In the security requirement assessment process, use cases will be classified with respect to the enriched security knowledge through heuristic findings. These heuristics will be described in this section. The automatically generated results of the classification and the heuristic findings will be passed to the requirements engineer, who is now able to enrich the existing security knowledge based on the findings of the security assessment. These findings now represent knowledge consisting of security-related terms that are extracted from the security assessment process and that are not part of the security knowledge base. The requirements engineer can now enhance the knowledge base with this information.

Classification of Words

In general, use cases are written in natural language. Therefore, we use natural language processing for their security assessment. Semantic similarity is defined as the similar meaning of two potentially syntactical different words [Sch94a]. We focus on nouns in the requirements and incidents. To identify the nouns, a statistical part-of-speech tagger is inevitable [PPM04]. If a security affiliation exists for these nouns, they will be assigned to the attribute system component, entry points, asset, and trust level of the security abstraction model. The modelled security knowledge supports the assignment of extracted words to the attributes.

The semantic similarity between nouns can be measured based on the structure and content of *WordNet*. In WordNet, the nouns are organised in hierarchies [Fel98]. We adapt the method of the lowest common subsume (LCS) [JC97]. The concept of LCS is a tree-like lexical taxonomy in which the similarity of words will be described by the shortest path between them in the tree. If the information content of the LCS is above a predefined threshold, the similarity between two words is very low. Otherwise, both words are semantically similar. To get the LCS of two words, the paths by using their hypernyms listed in WordNet will be derived.

Measurement of Similarity Between Security Abstraction Models

To identify the alignment of security, we utilise the Needleman-Wunsch algorithm [NW70]. The algorithm is originally used to determine the similarity of amino acid sequence of two proteins: All possible pairs of sequences could be represented as a two-dimensional array. The similarity of two sequences is represented as a pathway through the array. A smallest match when comparing a pair of amino acids can be used, one from each protein. The maximum match is defined as the largest number of amino acids of one protein that can be matched with those of another protein.

This comparison was transferred to the comparison of security abstraction models with use cases. To detect whether a use case is security relevant or not, all steps included will be compared to the collected security knowledge in the form of steps of a security abstraction model. For this assessment, the previous explained LCS method of semantic similarity is used. If the calculated LCS-value is above a given threshold, there is likely a vulnerability in a given use case. The results of every assessment are stored in a two-dimensional matrix, which is created for every security abstraction model comparison. The matrix cells contain LCS-values for the indication of similarity of two specific steps. In Table 5.3, an example of a comparison of a use case with the steps UC1 and UC2 and a security incident (SI) with the steps SI1 and SI2 are shown.

(Semi)-Automatic Acquisition of Tacit Knowledge

We interleaved the refinement and knowledge enrichment of the knowledge base in the security relevance assessment of use cases as an active learning mechanism. The requirement engineer actively decides to acquire potentially new security

Table 5.3 Extract of the comparison of two security abstraction models

		Security incident	
		Step SI1	Step SI2
Use case	Step UC1	0.5	1.5
	Step UC2	1.0	0.1

knowledge, such as the modification, reinforcement, and refinement of existing knowledge.

For this enrichment, there are two different results of classification, which process different information. The first are the true positives. In our approach, these are use cases that would be correctly classified as security-related. They enhance the knowledge through correctly classified terms. For example, if in the sentence "The user enters an identification number and a pin" the pin will be identified as security-related, we conclude via the existing linguistic dependency between pin and identification number that both are security-related. Besides classifying this sentence, the new insight can also be added as additional knowledge to the knowledge base.

The knowledge base can even be extended by false positives. They will be considered for specifying the terms for a certain domain. For this purpose, falsely classified scenario steps identified by the similarity computation concluding the attributes (system component, asset, entry point, and trust level) will be considered. If the value of similarity for an attribute is under a predefined threshold, there is an uncertainty for the classification. Therefore, the requirement engineer can actively manage whether a term should be excluded or included for the security classification approach. Afterwards, the learned security knowledge can be enriched by a security expert with additional security information (e.g. security standards and guidelines). Explicit security knowledge and precision grow over time.

5.3.4 Tacit Security Knowledge Examples

We applied our approach to the CoCoME case study. However, there is only a limited number of security-related requirements in CoCoME. Most of those had to be introduced for demonstrating the feasibility of our approach. Although intentionally inserted problems may be useful for concept demonstration, there are obvious threats to validity.

Therefore, we decided to strengthen the evaluation by using a second, larger example provided by others. The iTrust medical system case study is used by many researchers as a benchmark for security. Since it resembles CoCoME in many aspects, findings are relevant for the application domain represented by CoCoME.

In iTrust, a medical health care system [11], patients are able to manage their health records, such as medical items, and personnel can organise their work. If sensitive patient data are stored, only a limited number of people should be allowed to receive insights into this data. Therefore, security is inevitable to prevent access by intruders. Version 23 of iTrust consists of 55 use cases written in natural language, and the health care system is developed as web application. Our goal is to evaluate whether our approach can support requirements engineers through the security assessment of requirements.

Ten of the use cases of iTrust were selected as initial security knowledge for the requirement elicitation. These use cases distinguish themselves from each other

Table 5.4 Derived misuse cases of the iThrust system [Gär+14]

Concept	Individuals
MUC1	
Asset	Initial password, security key
Entry point	Email
Trust level	User
MUC2	
Asset	Address
Entry point	Address field, health record, view, display
Trust level	Patient, health care personnel

in such a way that they have at least one different actor and cover a different functionality of iTrust.

Unfortunately, there is no security incident documentation in iTrust. Nevertheless, a security incident can also be seen as a use case for the attacker, whereas for the requirements engineer it would be a misuse case (MUC). Therefore, we created misuse cases with respect to the ten initial use cases. In our example, an MUC represents the steps of a specific security incident, which is created based on known security incidents that occurred in the past.

A security ontology was set up on the use cases and misuse cases. The terms of the medical health care domain were considered to embed the domain-specific knowledge in our knowledge base. Furthermore, the individuals of the misuse cases listed in Table 5.4, like system components, assets, trust levels, and entry points, were added as well.

Through the analysis of use cases, we identified for use case 1 (UC1) and use case 6 (UC6) that they are ambivalent because there exists an misuse case, which malicious users follow to attack the use cases. UC1 describes the sending of the initial password for a user account, which is required to login to iTrust, to a user via email after the creation of the account by the medical personnel. This leads us to misuse case 1 (MUC1), in which a hacker intercepts the email and uses the password to have access to the iTrust system. This procedure is called *hijacking*. In UC6, the patient can manage their visits to health care professionals (e.g. doctors) and is able to see a list of health care professionals who have insights into their patient data. This leads to MUC2, in which the address fields of the patient view contain vulnerability that enables cross site scripting (XSS). XSS is one of the most dangerous vulnerabilities in web applications. An improper neutralisation of input enables XSS. An attacker is able to inject malicious browser-executable content into the patient view to steal sensitive data (e.g. medical identification number or password). The named misuse cases are listed in Table 5.4.

For the evaluation of our approach, we split the evaluation into two iterations. For the first iteration, we considered 35 use cases in addition to the ten initial use cases for the security assessment. The results of the heuristic approach were compared to the results of the manual and previously done requirements elicitation. Based on the true and false positives, security knowledge refinement was performed.

Table 5.5 Evaluation results [Gär+14]

		ACC	FPR	FNR
1st iteration (n = 44)				
Our approach	MUC1	0.90	0.10	0.00
	MUC2	0.64	0.55	0.15
Naïve Bayes	MUC1/2	0.61	0.00	0.89
SVM	MUC1/2	0.57	0.00	1.00
k-NN	MUC1/2	0.57	0.15	0.83
2nd iteration (n = 55)				
Our approach	MUC1	0.98	0.00	0.14
	MUC2	0.84	0.14	0.23
Naïve Bayes	MUC1/2	0.71	0.11	0.67
k-NN	MUC1/2	0.76	0.00	0.68

In the second iteration, the refined knowledge was used for the heuristic security assessment of use cases.

For evaluating our approach, we compared its performance to the results of Naïve Bayes, Support Vector Machine (SVM), and the k-nearest neighbors algorithm (k-NN). As quality facets, we considered the accuracy (ACC), false positive rate (FPR), and false negative rate (FNR). Under the ACC, we understand the degree of correctly classified use cases with respect to all of them. The false positive rate is defined by the number of falsely classified security-related use cases. Conversely, the false negative rate measures the falsely classified use cases as non-security-related.

In the training phase, the initial use cases are labelled while considering the misuse cases. If an misuse case is related to a use case, the use case is labelled as security-related. Otherwise, it is labelled as non-security-related. A low FNR implies that most of the use cases were found, which is desired. In the first iteration, SVMs has an FPR of 1.0, which means that no security-related use cases were found. Therefore, we were not able to refine knowledge based on heuristic findings. Thus, we did not consider SVMs for further iterations. Viewing the results of iteration 2, our approach in fact got ACC 0.98 for MUC1 and 0.84 for MUC2, as well as an FNR for MUC1 with 0.14 and for MUC2 with 0.23 as the best results. Only the FPR of MUC2 with 0.14 is higher than the other approaches. The results are listed in Table 5.5. Nevertheless, this fact is regardless for the context of requirements elicitation. It is required to find all of the security-related use cases, which is affected by the FNR. Afterwards a security expert can sort out false positives to achieve only security-related use cases.

5.4 Tacit Knowledge During Run Time

The requirement elicitation phase examines a software system in its completeness, striving for a complete and correct description. This is usually being done by developers through discussions with stakeholders, which are typically represented

by customers or the initiator of a software project. However, the end user of a system might pose different requirements. This is why requirement elicitation also includes interviews with the end users of a software system. In particular, user feedback is valuable in case it is collected during the run time of a system. Therefore, capturing knowledge about how a software system is utilised in the field carries valuable knowledge and represents an important aspect in software evolution.

The idea of collecting and improving software systems based on user feedback is further encouraged by recent activities that evolved under the umbrella of Continuous Software Engineering (CSE). At the core of its encompassing activities, continuous delivery allows to distribute software increments in short cycles to users, reducing the time between a developer's change in the form of a commit pushed to a software repository and the execution of its corresponding software artefact by an end user within the target environment [Bos14, FS17].

In the remainder of this section, we establish a perspective on tacit knowledge during run time. We present an approach on how tacit usage knowledge can be extracted from observed user behaviour. Therefore, we introduce the application domain, which we limit to users of mobile applications, and elaborate on the taxonomy of feedback, which we use synonymously to usage knowledge—referring to any knowledge that resulted from observing user behaviour. We conclude this section by describing preliminary results of a current research project.

5.4.1 Usage Knowledge in Software Evolution

Systems are designed by developers and their interpretation of how users utilise software, as described in Sect. 5.4.2. Information on users and on how they employ a system is rarely present during requirement elicitation. The feedback and behaviour of users reveal insights that help to evolve a long-living software system, in particular regarding the following shortcomings:

- Existing requirements are no longer applicable and need to be refined.
- New requirements are demanded that have not been considered during the initial phase of requirements elicitation.
- New user groups evolved, and users' intentions and requirements changed over time, which results in the need to adapt existing requirements.

Current software engineering practices apply iterative development processes that allow for the integration of users' feedback. This feedback can be divided into two groups [MHR09]: feedback that has been provided explicitly by the user—*conscious feedback*—and feedback that they provide indirectly and thereby implicitly—*unconscious feedback*—as an integral part of the application usage. Figure 5.2 depicts the taxonomy of conscious and unconscious feedback.

Conscious feedback is usually utilised during software evolution and is a rich source of usage knowledge. Users try to reach out to the developers, for example in form of an app store review, via mail, or through any other social media platform.

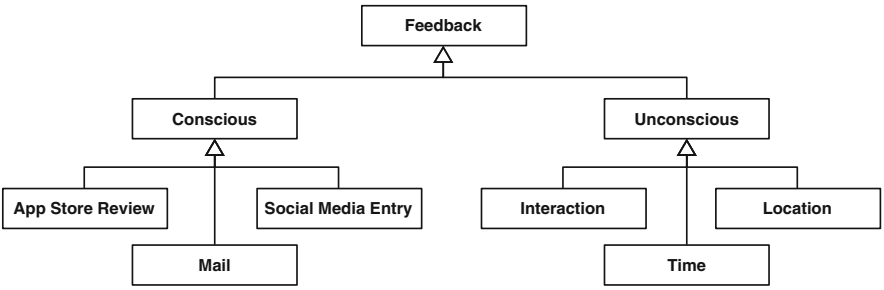

Fig. 5.2 A taxonomy of feedback provided by the end users of a mobile application

They relay their experience and clearly address a problem they had encountered. The utilisation of unconscious feedback requires a more advanced procedure of usage knowledge understanding since interactions, such as clicks or taps, or contextual information, such as time and location, need to be processed; well-adjusted methods can be used to retrieve such precise information about user interaction [Joh+18a].

It is unconscious feedback that enables the detection of tacit usage knowledge that can be utilised for software evolution. Tacit knowledge in the context of usage knowledge goes beyond user analytics. It describes the users' feelings, ideas, and insights about a software system that they are unable to express in conscious feedback such as written text. In particular, it is knowledge that they apply without knowing it, which might not follow the way the system was designed in the first place.

We want to inspect runtime tacit knowledge with a concrete example. Imagine a mobile application that offers users the possibility to read news articles that are presented in full screen. The developers implemented two possibilities to enable the navigation between entries: (a) using a swipe gesture or (b) selecting a dot at the bottom of the page that represents every news entry currently available. First-time users might only use the dots to navigate since it is the most obvious way. However, this is tedious since the dots are tiny and hard to spot. It is only until the moment the users discover the swipe gesture that they learn a new, more intuitive and convenient way of navigation. When developers understand the users' interaction with the application, they are able to react and either improve the button navigation or add a distinct introduction to the swipe navigation. Similarly, as an additional example, navigating through a vertical list of entries on a mobile device can be accomplished in different ways: Users may perform (a) a long-lasting, exaggerated gesture or (b) multiple precise, yet repetitive, short swipes to move through the list. While the first behaviour indicates easily readable content, the latter one might be interpreted in a way that the list's content is hardly understandable and requires the full attention of a user.

5.4.2 Modelling of Knowledge

In our following analysis, we assume four main entities as illustrated in Fig. 5.3: The *User* is the main actor who uses the *Application*, which represents a software system.

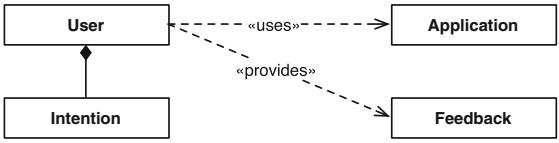

Fig. 5.3 Analysis of the application domain of usage knowledge

After and during the application usage, they provide *Feedback*, which can be further differentiated, as depicted in Fig. 5.2. The feedback is based on the user's *Intention*. This intention represents the user's idea of how they expect the application to accomplish a given task and how to behave given a certain interaction model posed by the application. To capture the intangible concepts of a user's intention, Norman introduced the *Conceptual Model*, which aims to formalise different perspectives on a similar issue [ND86]. He describes a relationship between the conceptual model and the *Mental Model* of every stakeholder that interacts with an application. In Fig. 5.4, we sketch involved models, in which the actual reality is represented by the conceptual model, which might be interpreted in different ways.

According to Norman, two models are derived from the conceptual model: a *System Model* and a mental model. Both of them are related to and formed by a variety of models. The system model is the result of the discussions of domain experts while applying domain knowledge. They create artefacts—the software increments—in accordance with their understanding of a design, functional, and object model. The mental model encompasses the users' perception on that specific artefact. The mental model depends on educational, cultural, or other general knowledge models that can be summarised under tacit knowledge. Eventually, it is the *User Interface* that brings both models together. The overall goal is to achieve

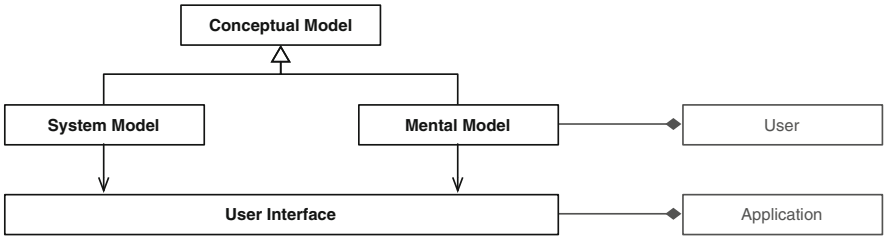

Fig. 5.4 Interpretation of the conceptual model as a system model and mental model [ND86] and their combination with the application domain coloured in grey on the right part of the figure

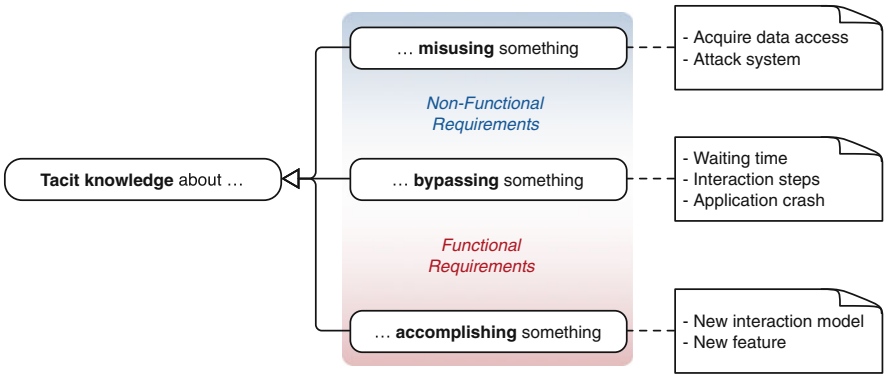

Fig. 5.5 Categories of tacit knowledge in the context of software evolution

a natural mapping [Nor13] that is characterised by a minimal amount of model disagreement. The disagreement of the system model and the mental model might be measured in the user's interaction with the application's user interface, which is built from the developer's system model.

The user's intention is closely bound to the tacit knowledge that users are unable to describe. As shown in Fig. 5.5, tacit knowledge can be categorised into three groups, though there might be more ways of distinguishing tacit knowledge in software evolution.

5.4.3 Identification and Extraction of Tacit Usage Knowledge

For the identification and extraction of tacit usage knowledge during run time, we propose a semi-automated approach. First, the occurrence of potential tacit knowledge needs to be detected, which should be accomplished using machine learning—we introduce the concept of runtime personas for this purpose in the next section. Second, in the event of tacit knowledge detection, a request for more qualitative feedback is posted. Third, a manual step of integrating the detected situation of tacit knowledge with the qualitative feedback of users is performed. Steps 2 and 3 are described as the extraction of tacit usage knowledge in the last part of this section.

Runtime Personas

Tacit knowledge needs to evolve; it is not existent at the moment a user starts using a software system. It develops over time, figuratively, though the temporal aspect can be part of the consideration. Other characteristics that indicate the familiarity of

a user with the system might be clicks or any other quantifiable value summarized under unconscious feedback.

We apply Polanyi's terminology and assumptions for describing the identification of tacit knowledge. He introduces the *proximal* term and the *distal* term [PS09]. Proximal terms are considered the origin of an action, a starting point, or any event, such as interaction with the user interface, that eventually leads to a result. Such a result is described by the distal term, which can be any end point, intention, or goal, such as the execution of a software feature. During the first step of our approach, we aim to identify the connection between these two terms, which previously remained tacit. Polanyi states that there is a fluctuating link between the two of them, which eventually ends in a bold, established relationship—the tacit knowledge. Figure 5.6 illustrates this evolvement of tacit knowledge separated over a time span of different observations during the run time of a software system.

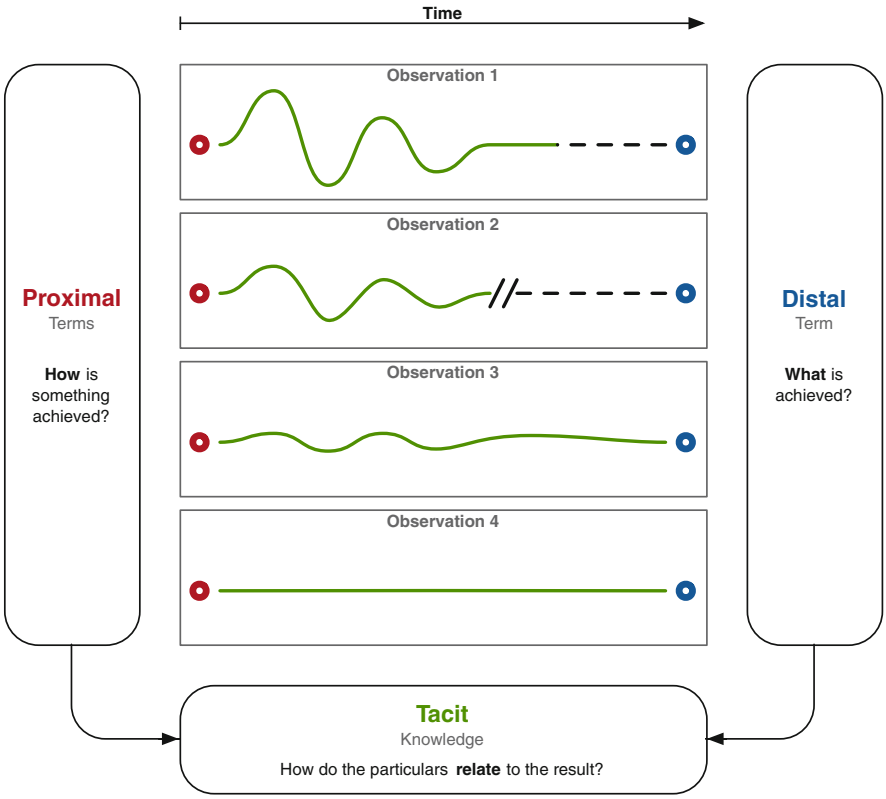

Fig. 5.6 The relation between the proximal terms (red), distal term (blue), and tacit knowledge (green). The proximal terms, for example taps by the users, are eventually mapped to the distal term, for example the feature execution. According to Polanyi [PS09], tacit knowledge can be understood as the established mapping between the proximal and distal terms

Polanyi argues that "we are aware of the proximal term of an act of tacit knowing in the appearance of its distal term" [PS09] and continues to define this finding as the *phenomenal* structure [PS09]. Consequently, if we identified the distal term of a given tacit knowledge while users established a mature connection between both terms, we might be able to derive the proximal terms that are of great value to start understanding software usage. For the purpose of collecting and allocating observations, we introduce the concept of **Runtime Personas**. According to Polanyi, tacit knowledge is a person-related concept, which matches the persona definition. Run time personas form a container to capture the evolvement and eventually the discovery of tacit knowledge. The evolvement of a run time persona is initiated with traditional personas as a first, optimal representation of tacit knowledge, while it gets enriched throughout multiple stages and new findings.

This process can be understood best by giving an example aligned with Fig. 5.6. The distal term defines the results of an action—the consequence or outcome, depending on the observation. In the context of software engineering, this could be the execution of a feature, in particular related interactions with the user interface. Referring to our initial examples, the distal term could be expressed in consuming a list of information on a mobile application. It is the proximal terms, the *particulars*, that a user may not be able to tell when using the software system. In our example, it manifests itself in the way the user interacts with the list to traverse the list's content. The challenge lies in the discovery of the connection between this interaction and the usage of the list, namely its corresponding distal term. Traditional personas [Coo99] serve as the starting point. They describe a person's characteristics that qualify them for the usage of a feature, in particular reaching the previously defined distal term. Further, they encapsulate the observations resulting from the asymptotic process of information extracting. For instance, *Observation 1* refers to a situation in which it is not clear if a user's interaction leads to a feature usage. *Observation 2* seems promising, but indications stopped before it could be clearly mapped to the usage of the feature. *Observation 3* represents the first time that a definite correlation between the proximal and distal terms could be established, while it still includes some fluctuations. Finally, *Observation 4* encompasses a clear link between the two terms, allowing for the derivation of tacit knowledge.

Extraction of Tacit Usage Knowledge

Adapting Polanyi's hypothesis of the phenomenal structure of tacit knowledge to the context of usage knowledge in software systems, users are aware of their interactions from which they are attending to accomplish the feature—in appearance of that specific feature. This allows for extracting the tacit knowledge in the event of *Observation 4*.

We propose utilising a modal window that asks the user for qualitative feedback, as shown in Fig. 5.7. It is triggered as soon as a distinct relation between proximal and distal terms is detected. In particular, we imagine gaining insights with regard to the following questions:

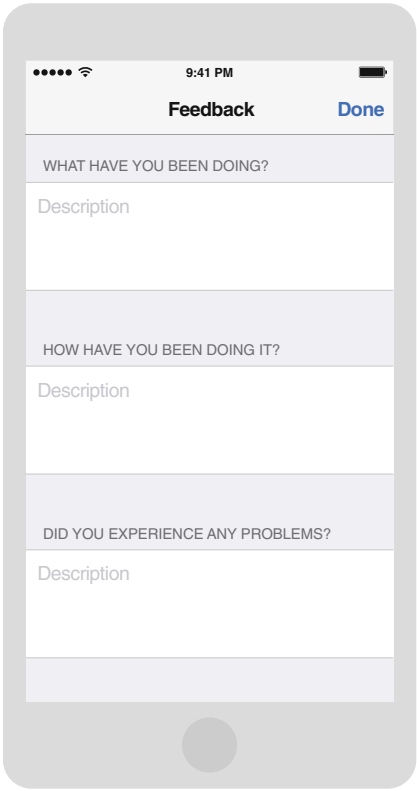

Fig. 5.7 Mockup of requesting feedback from user

- What has the user been trying to do?
- How did the user try to achieve it?
- Did the user experience any problems during this process?

The qualitative feedback enables the developer to understand and externalise the tacit knowledge carried out by the users during run time. For integrating the usage observation with the qualitative feedback, we propose the introduction of a dashboard [Joh+17b]. The dashboard is a central component of the **CURES** project. Within this dashboard, we envision to visually display categories of equivalence classes—either based on the usage knowledge or by groups of distal terms, namely the performed features. This allows the developers to augment information from multiple feedback and find an optimal solution for integrating the new findings [Joh+17a].

A further extension to encourage users to provide more detailed information about the performed action could include a predefined selection of features—the distal term—potentially involved in the process. However, this would require the possibility to make a distinction in features used, in particular features offered by the software system.

Besides the integration and utilisation of the tacit runtime knowledge during design time tasks by the developer, recurring patterns of tacit knowledge can be caught during run time and utilised by the developer. For example, referring to Fig. 5.5, in case a user wants to bypass several process steps that in general cannot be removed from the application, the system could still provide a shortcut functionality as soon as this situation is detected.

5.4.4 Tacit Usage Knowledge Examples

We focus on the automatic creation of run time personas from usage behaviour within mobile applications. In this section, we describe examples of tacit usage knowledge from a current research project [Frö18].

We prepared an open-source mobile application with several modifications to record explicit usage data, such as interactions with the user interface in the form of taps and gestures, as well as other sensor data, such as gyroscope and tap pressure. We designed a catalogue of tasks to stimulate interaction within the application; for example, we asked to use a specific functionality of the application or to find out particular information that required them to navigate through several views of the application. The tasks were carefully chosen to encompass typical routines of user interactions, as well as aspects that allow to recognise the users' behaviour in unexpected situations. Based on the task execution by more than 100 individuals, we trained multiple classifiers to derive the following characteristics for our run time personas.

- *Person-related information* aims to characterise attributes that are highly individual to users, such as age groups distinguished by age ranges or their proficiency and skills in dealing with mobile applications, distinguished in beginner and expert groups.
- *Application-related information* aims to define the user's familiarity with the application at hand. This is reflected in attributes such as the familiarity level, while we distinguish between a beginner and expert level, and their mental phase with respect to the application usage, that is if they are exploring the interface or if they are productively working and interacting with its functionality.
- *Application-related usability issues* aim to reflect users' behaviour given a situation in which they encounter an unexpected system behaviour, such as inconsistencies in the user interface or missing user interaction elements in the user interface.

So far, based on the current evaluation of usage data, we receive good results on detecting situations in which users encounter an application-related usability issue. We hypothesise that this is based on the fact that they only occur during a short period of time, which is revealed in a distinct set of obvious changes in user behaviour. The exact characteristics of the features remain yet unknown. Equally promising results can be reported for detecting application-related information.

Here, the detection of the productivity status of a user results in especially good results. However, we assume a systematic error in measuring the productivity of a user. Currently, we label a behaviour as productive in case the user is *on their way* to using a functionality. In case they are *moving away from it*, for example navigating towards a view that is not related, leaving no further space to use the functionality, we consider their status as exploring. The results of the classifier depend on a threshold for distinguishing between these two states. Measurements for person-related information highly depend on the splitting of both the test and training data for the individual classifiers.

We observe that application-related information can be suitable for automatically deriving run time persona attributes. We hypothesise that this is because of their inherent semantic relation to the user interaction. Person-related information, on the other hand, is more challenging in its extraction and consequently is less accurate to detect. Multiple reasons for its low predictability might be found in the way of data collection.

In general, the presented approach to collect run time personas' characteristics and the resulting classifiers need to be treated with caution. Firstly and most importantly, the approach would highly benefit from even more individuals who provide usage data. In our model under consideration, we have an unbalanced distribution of person-related information. This could be the reasons why the models for person-related information might perform worse than the application-related information. We also acknowledge a high bias of the sample application that was used to collect the usage data. We tried to minimise this effect by tailoring the task scenarios around general user interface interactions that are typical for a majority of mobile applications. Overall, we suppose that several machine learning features for training a model remain undiscovered. Therefore, future research is required to find more machine learning features that reveal the main behaviour characteristics of an action.

5.5 Related Work

Tacit knowledge is present during various aspects of software evolution. For instance, it has been shown that developers share important rationale through chat messages to perform development tasks [Alk+17a, Alk+17b]. This observation fosters our assumption that there is more knowledge in existing artefacts that has not yet been externalised. In particular, LaToza et al. highlight knowledge that resides in developers' minds regarding the application of tools and activities to perform code tasks during software development [LVD06]. This chapter sets the focus on tacit knowledge to improve requirements elicitation by capturing additional information during the design and run time of a software system. In the following, we present existing work.

AlHogail and Berri [AB12] propose the development of architecture to preserve security knowledge within an organisation. They plan to perceive and distribute

security knowledge to tackle the problem of availability of security experts in software projects. This enables a faster reaction on security incidents. To preserve security knowledge, a template is used. Tsoumas and Gritzalis [TG06] present a security management approach for information systems containing security knowledge of different sources. Several approaches deal with the management of security knowledge in ontologies [Ras+01, KR07, BKK05]. Lee et al. [Lee+06] introduce an approach for the extraction of relevant concepts from documents to build a problem domain ontology. Jung et al. [JHS99] developed a reasoning approach to use past security accidents in the risk analysis of e-commerce systems. To apply this approach, the problem must be formatted into a specific case representation, which makes additional effort necessary.

Most of the approaches are not considering the evolution of security knowledge intensively. The support of requirements engineers who use past events from gathered security knowledge in the context of requirements elicitation was not taken into account in most approaches. Furthermore, cases in which knowledge changes over time were also not considered.

When switching the perspective from a software architect or requirement analyst to end users, for example the users of a software system, the runtime aspects of a software system provide a rich source of tacit knowledge. Following Roehm et al.'s findings, developers try to make use of this by putting themselves in the shoes of users to understand program behaviour and get first ideas to further act on it [Roe+12].

By applying a semi-automatic approach, Damevski et al. mine large-scale datasets of IDE interactions [Dam+17]. Therefore, they aim to identify inefficient applications of IDE usage patterns relying on their observations of developers' activities during their daily development tasks. They begin with an automated approach that—after preparing the input data—encompasses a sequential pattern mining and filtering activity. Hereafter, clusters are created to determine common workflows of developers, which are verified by the authors and a developer survey. The approach of Damevski et al. shares a concrete process model to derive knowledge from usage behaviour for the specific domain of integrated development environments (IDEs). Our approach presented in Sect. 5.4 reflects the core idea of the approach presented by Damevski et al. In particular, we try to identify common usage patterns of a software increment.

Zhang et al. present a *quantitative bottom-up data-driven approach to create personas* in their paper *Data-Driven Personas: Constructing Archetypal Users with Clickstreams and User Telemetry* [ZBS16]. Their approach on creating personas solely relies on click streams, while we want to incorporate other data as well, such as the location or any meta data that describes *how* and *when* clicks occurred in order to provide additional semantics.

Almeida et al. acknowledge the presence of poorly designed applications that prevent users from using them and sustainable maintenance and evolution [Alm+15]. They introduce a usability smell catalogue that allows for their identification, as well as refactoring the problems in question. Similarly, we strive to find *behavioural smells* that provide information about the users [Joh18].

Gadler et al. apply log mining to derive the use of a system; utilising Hidden Markov Models, they automatically represent user's intention [Gad+17]. We want to apply a similar approach to understand the users' intention when interacting with a new software increment.

5.6 Conclusion

To conclude this chapter, we provide a brief summary of tacit knowledge in software evolution, an outlook on future challenges, as well as further reading suggestions.

5.6.1 Summary

We described two approaches to identify and extract tacit knowledge during the design time and run time of software systems. During the development of the approaches introduced in Sects. 5.3 and 5.4, we encountered various lessons learned, which we summarise subsequently.

We acknowledge that requirements which become new features might be relevant for security. Identifying tacit knowledge in the form of security knowledge is a difficult task for which a good understanding of security and the domain of the software is necessary. Nature language processing can support the requirement engineer during this task.

Extracting tacit usage knowledge during run time raises various challenges. As indicated in Fig. 5.6, potentially *wrong* usage behaviour might eventually transition into a pattern that is of interest and relevant for a new feature or functionality of an application. This learning phase needs to be a core element in the detection of usage behaviour, making it an important reference that points to the tacit knowledge. Likewise, it is important to distinguish tacit knowledge from any kind of *noise* effects. Eventually, we learned that only a limited set of new features can be detected, while the quality of insights highly depends on the application in question.

Further discussions on security and its maintenance are described in Chap. 9. Linking the tacit usage knowledge to other knowledge types, such as decision knowledge described in Chap. 6, provides new possibilities to further support software evolution.

5.6.2 Outlook

Tacit knowledge in the domain of software evolution promises future research areas to improve processes and software quality. In the following, we elaborate on

multiple aspects of design and runtime tacit knowledge that we propose to continue to work on in the future.

We developed an approach to identify security-related requirements semi-automatically using natural language processing. The success of our approach depends on the quality of security knowledge. Detailed knowledge leads to a more helpful base of security knowledge for our approach. One challenge is to retrieve and model the security knowledge to make it accessible for further requirement elicitation. Our approach can identify vulnerabilities in requirements written in natural language based on security incidents. With the iThrust case study, we have shown that our approach performs better than other approaches, such as Naïve Bayes, k-NN, and SVMs. To apply our approach in an industrial setting, we have to evaluate the level of detail that is used to document security incidents. Furthermore, we need to investigate if intermediate feedback on security issues in requirements improves the elicitation of security requirements.

A general, major challenge for future research efforts regarding runtime tacit knowledge will be the detection of deviations between explicitly elicited requirements and implicitly derived requirements based on users' behaviour. In particular, creating a traceability link between these requirement sources still poses a challenge in the exploration of tacit usage knowledge.

Two additional challenges should be investigated to further evolve software engineering regarding tacit knowledge during run time. We found a challenge in detecting actual error conditions. In particular, this requires to decide whether a behavioural pattern or sequence is relevant or if it is simply *noise*, which is irrelevant for the evaluation (see *Observation 1* and *Observation 2* in Fig. 5.6). This challenge results in a fundamental question: Is every behaviour relevant and is there such a thing as noise? Furthermore, the actual interaction with users, as described in Sect. 5.4.3, needs to be clearly defined. This includes the question on when a user can be interrupted in order to retrieve their state, that is what they have been doing, how they were doing it, and if they experienced any problems (see Fig. 5.7). We identified two requirements that need to be fulfilled to spot the appropriate moment to interrupt a user and thereby prevent negative interruptions. First, a user should only be interrupted if it can be guaranteed that it will not interfere with their current workflow. Second, no critical process should be disturbed. Both requirements, however, pose new challenges. A balance needs to be found to keep a minimal time span between the interaction and the interruption. A delay, though, results in the problem that traceability should be guaranteed; that is, the users' feedback should be allocated clearly to an interaction. We envision to develop a tacit knowledge characteristic similar to the properties defined in database transactions: atomicity, consistency, isolation, durability [HR83].

5.6.3 Further Reading

In the project SecVolution, Bürger et al. presented a framework that analyses the environment, security-related requirements, and observations to provide an automated reaction to observed changes and to ensure a certain security level for long-living information systems [Bür+18].

As part of our previous work, we developed a prototype called FOCUS for the documentation of non-functional requirements while using execution traces, as well as video screencasts underlined by audio comments [Sch06]. The documentation can be created as a by-product via recording the application of a task [Sch06].One field for using our tool is security. Therefore, we have enhanced this documentation by a semi-automated approach to analyse security vulnerabilities based on remote code exploits for Java applications [VKK17]. The analysis enables the localisation of a source code vulnerability while distinguishing a penetration test recording with a recording of the regular behaviour of the same application. Gärtner et al. developed a tool-based approach, which provides heuristic feedback on security-related aspects of requirements to document decisions [Gär+14]. For this purpose, a decision model is used to systematically capture and document requirements, design decisions, as well as related rationale.

Pagano and Roehm described the difference between expected and observed user behaviour based on different perceptions of the conceptual model [Pag13, Roe15], an aspect that we address in Sect. 5.4.2. Roehm et al. investigated derivations in the descriptions of use cases with observed behaviour of users by applying machine learning techniques [Roe+13a]. The interaction with user interface elements is investigated by Roehm et al. using an approach to associate user interactions with application bugs to enable failure reproduction [Roe+13b].

We provide and maintain the source code and further explanation of tools and platforms for usage knowledge understanding in an online repository.[1]

[1] https://github.com/cures-hub.