Chapter 15
Experience-Based Requirements Engineering Tools

E. Knauss and S. Meyer

Abstract Writing a good software requirement specification is a complex task. Many different aspects must be taken into account; most of them can only be learned through experience. Being aware of experiences and distilled best practices at the right time when writing a specification is another challenge. Experience-based requirements engineering tools make sharing and reuse of experience feasible. In this chapter, we present design principles for such tools, define a learning model to describe how organisations and individuals can learn new experiences by using them, and sketch a strategy for evaluating experience-based requirements engineering tools. We highlight these concepts with an example.

15.1 Introduction

Authors of software requirement specifications (SRS) have to adopt the increasing complexity of today's software systems. This leads to more comprehensive and complicated requirements. In these situations, authors find it increasingly difficult to avoid ambiguities and contradictions in order to create a high-quality SRS.

There are many guidelines and best practices about how to create a good specification. Authors are expected to know them and to be able to apply them when needed. This leads to an information overflow, especially if the author is not a full-time requirements analyst. Furthermore, writing good requirements needs a lot of experience. Gaining and sharing experiences in writing SRS is a challenging task. In general, software developing organisations try to address these challenges by systematic knowledge and experience management [1], thus becoming Learning Software Organisations (LSO) [2]. However, there is limited support for organisational learning in requirements engineering.

E. Knauss (✉) • S. Meyer
Leibniz Universität Hannover, Hanover, Germany
e-mail: eric.knauss@inf.uni-hannover.de; sebastian.meyer@inf.uni-hannover.de

When trying to support analysts in this situation, one has to face two problems that are intertwined with each other: On the one hand, an author of a SRS is not aware of guidelines or best practices that could help to create a good specification. On the other hand, experienced requirements engineers that know these best practices do not know how to share them with others in a form that supports efficient reuse.

In this chapter, we discuss how tools can address these two problems and offer experience-based support. Based on a learning model, we address the following research question:

> Can experience-based tools support learning requirements engineering and do they lead to better requirements documentation?

We describe important design principles for such tools in Sect. 15.2 and illustrate them at the hand of the example of our HeRA tool (Sect. 15.3). In Sect. 15.4, we relate other works to these concepts. In addition, we discuss how our concepts can be applied and evaluated in Sect. 15.5. Our concepts are specific to requirements engineering activities in allowing dealing with inconsistent and incomplete documents with limited structure and formality, that is, the kind of documents typically encountered during the requirements analysis.

15.2 Design Principles

This section gives design principles for requirements engineering tools based on principles of experience and knowledge management. Based on Schneider, we define experience as a specific kind of knowledge being acquired by a person by being involved [1]:

Definition 1. *Experience* (*Schneider* [1]). An experience is defined as a three-tuple consisting of:

(a) An observation
(b) An emotion (with respect to the observed event)
(c) A conclusion or hypothesis (derived from the observed event and emotion)

We call a tool that supports requirements engineering based on externalised experience an *experience-based requirements tool.*

Definition 2. *Experience-based requirements tool.* An experience-based requirements tool supports one or more of the following aspects of organisational learning during requirements engineering activities:

(a) *Learning*: the creation of new experience
(b) *Evaluation*: the evaluation of existing experience
(c) *Management*: the distribution, updating, and refinement of existing experience
(d) *Application*: the usage of existing experience in a given context

An example of a simple experience-based requirements engineering tool is a tool allowing tailoring (and *manage*) requirements templates. Based on experience, the template can be improved. A better template (i.e. tailored to a specific project context) supports requirements engineering. More sophisticated examples include automatic requirements checkers, because they support the *application* of existing experience about quality problems that should be removed from requirements documentation.

Often, the discussion of tools that automatically analyse requirements documentation is limited to the discussion of their recall and precision (e.g. in [3]). Here we give a broader model for requirements analysis tools allowing us to describe their usefulness for supporting continuous improvement and organisational learning in requirements engineering activities.

Kiyavitskaya et al. argue that tools used for identifying problems in requirements documentation should have 100 % recall, that is, they should find all problematic requirements [3]. Only then, requirements analysts can focus on reading only the problematic requirements returned by the tool. With lower recall, analysts have to read all requirements again to find all problems. In addition, many false positives have a similar effect: If the tool reports almost every requirement to be problematic, the analyst does not gain much. Thus, a high precision is also important. We agree that recall and precision are important properties and subsume them in the property *reliability*. Note that even low reliability might be acceptable, if other properties of the tool add enough value.

Definition 3. *Reliability*. The reliability of a requirements checking tool is defined inversely proportional to the number of *type one errors* (the checking tool reports a problem, but there is none; metric: precision) and *type two errors* (the checking tool reports no problem, but there is one; metric: recall). The reliability can be:

(a) *Low:* Recommendations of tool are often incorrect.
(b) *Medium:* F-measure > 0.7 (f-measure is a combined metric based on the harmonic mean of precision and recall).
(c) *High:* F-measure > 0.85.

Especially, when focusing on learning aspects of such tools, it is important to switch the perspective and analyse how such tools affect the work of requirements analysts. For this, we add another definition: *authority* of tools.

Definition 4. *Authority*. The degree to that users rely on feedback of automatic requirements checkers. The authority can be:

(a) *Low*: Recommendations of tool are seldom adhered to.
(b) *Medium:* Recommendations of tool are more often adhered to than not.
(c) *High:* Recommendations of tool are mandatory.

This definition has two facets:

1. In an extreme case, the requirements engineering workflow of an organisation could dictate analysts to react on all findings from an experience-based requirements engineering tool, no matter whether the finding stands for an actual problem or not. This makes sense, if changing 100 requirements without problems

is cheaper than missing one requirement with a problem. We conclude: *Authority* can be independent from *reliability*.

2. Otherwise, the experience-based tool has to earn authority by being useful. We conclude: *Reliability* (as expressed by precision and recall) has a strong impact on usefulness.

Note that inexperienced users could rely on bad feedback. Therefore, it is important to distinguish between the concepts of reliability and authority. Beyond these concepts, the usefulness of computer-based feedback is strongly influenced by the *proactivity* and by the *degree of interpretation*.

Definition 5. *Proactivity*. The degree to that an automatic requirements checker defines the time of feedback. The proactivity can be:

(a) *Low (=reactive):* Feedback is only given if the user requests it.
(b) *Medium:* Feedback is triggered by the user's specific actions.
(c) *High:* Time of feedback is determined based on more complex rules.

Proactivity defines the trigger of the feedback. Often, it is easier to directly improve a problematic requirement. When using a reactive tool, the analyst needs to understand that feedback will be useful in a given situation. Because of the high time pressure during requirements analysis, this often leads to the problem that feedback is only given very late. Then, it is much more difficult to repair problematic requirements: Other requirements might depend on the problematic one. In addition, the analyst needs to understand the exact circumstances of a given requirement.

Definition 6. *Degree of interpretation.* The degree to that an automatic requirements checker gathers and interprets data about a given situation beyond concrete available data. The degree of interpretation can be:

(a) *Low:* The tool reproduces existing data and lets the user interpret it.
(b) *Medium:* The tool sorts and filters data, thus suggesting a specific interpretation.
(c) *High:* The tool refines and interprets data.

Finally, an important property of experience-based tools is the ability to learn.

Definition 7. *Learning ability*. The ability to integrate new experiences or refine existing experiences in the experience-based requirements tool. Learning ability can be:

(a) *Low:* New experience has to be hard coded into the tool by experts.
(b) *Medium:* Special facilities exist in the tool to allow users to add new experience.
(c) *High:* The tool can automatically adopt, for example, based on observing its user.

The learning ability is important to allow adopting the experience-based tool to a specific situation. Furthermore, the learning ability is the primary facility to support continuous improvement and organisational learning: If new knowledge is encoded into the experience-based tool, the organisation owning the experience-based tool has learnt.

In requirements engineering, we encounter a large variety of document types. We aim to define our design principles independent from the specific document type, but of course the properties of documents handled by a tool have a certain impact on the performance of the tool. For our purposes, it is sufficient to distinguish two properties of requirements documents: (a) *formality* and (b) *maturity*. The more formal the syntax of a requirements document (or model) is, the easier it is to analyse for requirements checkers. Formal requirements models even allow simulation and other mechanisms for quality insurance. However, in many projects, requirements are specified without too much formalism, especially in the beginning of a project. Experience-based tools that offer support early during requirements engineering activities should be able to handle informal requirements descriptions as well as incomplete documents (with low maturity).

15.2.1 Learning Through Experience: Heuristic Critiques

In this section, we describe how the presented design principles affect learning and experience management. First, we introduce the concept of *heuristic critiques*:

Definition 8. *Heuristic critique*. Computer based feedback to an activity or work product (e.g. requirements documentation) based on experience. A heuristic critique consists of:

(a) A *heuristic rule* that can be evaluated by a computer
(b) A *criticality*
(c) A meaningful and constructive *message*

A heuristic critique depicts a single automatic requirements check. Furthermore, it supports Learning Software Organisations (LSO). Such an LSO focuses on developing software while supporting the following aspects of organisational learning [1]:

(a) Learning of the individuals in the organisation
(b) Organisation-wide collection of knowledge and experience
(c) Cultivation of an organisation-wide infrastructure for exchanging knowledge and experience

When integrated in requirements engineering tools, heuristic critiques offer support for a LSO. Table 15.1 gives an example of the relation between heuristic critique and experience (see Definition 1). While implementing a requirement, a developer makes an experience (left hand of Table 15.1). A heuristic could be created that covers part of this experience (right hand of Table 15.1).

The heuristic rule gives a good solution without guarantee for optimality or feasibility. In the example, passive voice can be found by computers and often (not always) leads to the detection of unclear responsibilities. Examples of other heuristic critiques include:

- *Inconsistencies:* If two similar requirements (e.g. use cases with similar title) are detected, give a warning and suggest that the user merges both requirements or clarifies them to avoid inconsistencies.

Table 15.1 Example: a heuristic critique encodes an experience (cf. [33])

Part of exp.	Experience	Heuristic critique	*Part of heur. crit.*
(*i*) *Observation*	Req. was misunderstood, because we had not specified who was responsible	If passive voice is detected…	(*a*) *Heuristic rule*
(*ii*) *Emotion*	It took a week to rework the module – just for sloppy writing!	…give a warning	(*b*) *Notion of severity*
(*iii*) *Conclusion/hypothesis*	It should always be spelled out who is responsible for an action. Avoid passive voice!	…asking the user to use active voice and state responsibility	(*c*) *Meaningful and constructive msg.*

- *Ambiguities:* If a weak word (e.g. always, sometimes) is encountered, give a warning and ask the user to specify the circumstances of the requirement more exactly.
- *Incompleteness:* If a use case on user goal level is found that is not included or does not extend a use case on business level, give a warning and ask the user to add the missing link or to specify the missing business goal.

15.2.2 Learning Model

Figure 15.1 shows two important areas of learning, supported by heuristic critiques. Learning occurs on individual and organisational level during the activity of writing requirements.

15.2.2.1 Individual Learning: Reflect and Apply

Under pressure, analysts write bad requirements, even if they know how to do it better. It is good to have a mentor who gives gentle reminders until knowledge develops to skills. Heuristic critiques like in Table 15.1 could give just this type of feedback.

Reflect: If a heuristic detects passive voice in a requirement and fires a warning, the requirements engineer is interrupted in his task. This gives him the chance to reflect about the action currently taken out; a breakdown occurs [4, 5]. This facilitates learning through reflection, if evaluated directly on the requirements engineers' input. Note that we depict this information flow as irreproducible, because the reflexion depends not only on the heuristic critique but also on the mental context and state of mind of the analyst.

Apply: The requirements engineer might already know how to write good requirements in general. Nevertheless, passive sentences may slip into a specification

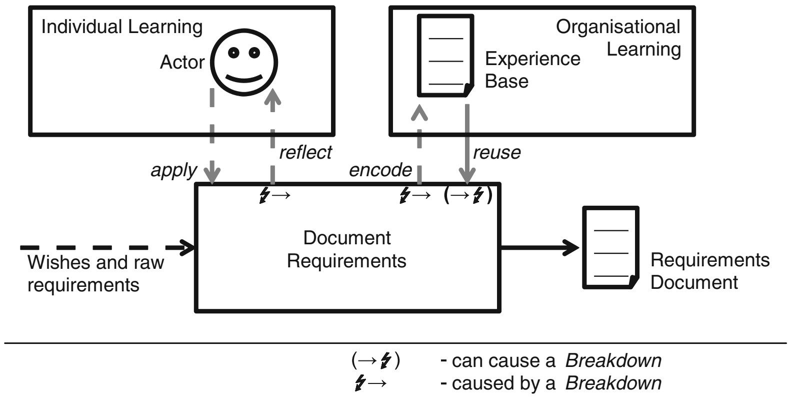


Fig. 15.1 Learning model (cf. [32, 33]): information flows (*arrows*; *dashed*, irreproducible; *grey*, experiences) are the foundation of *individual* and *organisational learning*. *Heuristic critiques* can stimulate the information flows by causing *breakdowns*

during periods of intense writing. Reminders and warnings help to apply and repeat the knowledge. Thus, they support turning abstract knowledge into internalised skills and help writing good requirements. Again, this information flow is irreproducible, as the analyst might choose different solutions, even if he makes the same mistake again.

15.2.2.2 Organisational Learning: Reuse and Encode

Heuristic critiques allow codifying experiences in a useful way.

Reuse: Based on heuristics, computers can be used to find situations matching the observation that led to the original experience. A more or less disruptive message points to potential improvements that are inferred from reported emotions.

Encode: Heuristic warnings are not always correct, for example, an actor could be specified even in a passive sentence. Furthermore, they are not always applicable. For a condition, which is stated in requirements documentation, use of passive voice is unproblematic. If such a situation is observed during a breakdown, the requirements engineer can refine the heuristic warning and specify that it should not be applied to conditions. Thus, experience is added to the organisations' knowledge base. As a by-product, the growing body of codified experience adopts a manageable granularity for a LSO's knowledge base. This information flow is irreproducible, as solutions differ based on the context that leaded to the encoding of the new experience.

15.2.3 Research Questions

We derive our research questions from the learning model in Fig. 15.1. Each (group of) experience and knowledge flow leads to a specific question:

- *RQ 1*. Can experience-based tools support individual learning? We need to investigate the impact on individuals (*a) applying* and (*b) reflecting* as depicted in Fig. 15.1.

- *RQ 2*. Can experience-based tools support organisational learning? We need to investigate organisation-wide (*a) reuse* and (*b) encoding* of experience (c.f. Fig. 15.1.).
- *RQ 3*. Do experience-based tools lead to better requirements documentation? We need to investigate if (*a) quality* and (*b) costs* of requirements documentation are improved with experience-based tools.

15.3 Example: The Heuristic Requirements Assistant (HeRA)

This section gives an example of how to integrate the described design principles into an experience-based requirements engineering tool. While we demonstrate the principles and their implementation on our own HeRA tool, there are of course other tools. We give a short overview of them at the end of this section.

HeRA is our implementation of an experience-based requirements engineering tool. It is basically a smart requirements editor with several heuristic feedback facilities [6]. While HeRA is designed to be extensible, we focus in HeRA's original critique system for describing the key concepts.

HeRA is based on Fischer's architecture for domain-oriented design environments (DODE) [5]. The central part of this architecture is a construction component. In the case of HeRA, requirements are "constructed" using a general-purpose requirements editor, a use case editor, and a glossary editor. These editors allow constructing specific artefacts (i.e. requirements, use cases, and a glossary). HeRA offers two other DODE components, namely, the argumentation component and the simulation component. Figure 15.2 shows the structure of the HeRA tool.

We describe the interplay of construction, argumentation, and simulation components based on use cases. Use cases are used to describe the interactions of a user with the system to construct (based on templates suggested by Cockburn [7]). HeRA was designed to support the requirements engineer with heuristic feedback: It analyses the input and warns the user if it detects ambiguities or incomplete specifications. Furthermore, HeRA can generate diagrams like use case diagrams from the Unified Modelling Language (UML) or event-driven process chain (EPC) models that show how the current user goals relate to the business goal or the global process. If needed, a glossary assistant can be used to ensure consistent use of important terms. On demand, HeRA computes use case points and displays an effort estimation associated with the use cases. Use case points are calculated based on the actors and transaction between them. The HeRA module implements the use case points method as described by Kusumoto et al. [8]. All of these perspectives are derived while the use case is being written. In this way, the author gets immediate feedback on the input and may improve it. We have successfully applied HeRA during interviews and workshops (see Sect. 5). Supported by HeRA's feedback facilities, we aim to achieve the following:

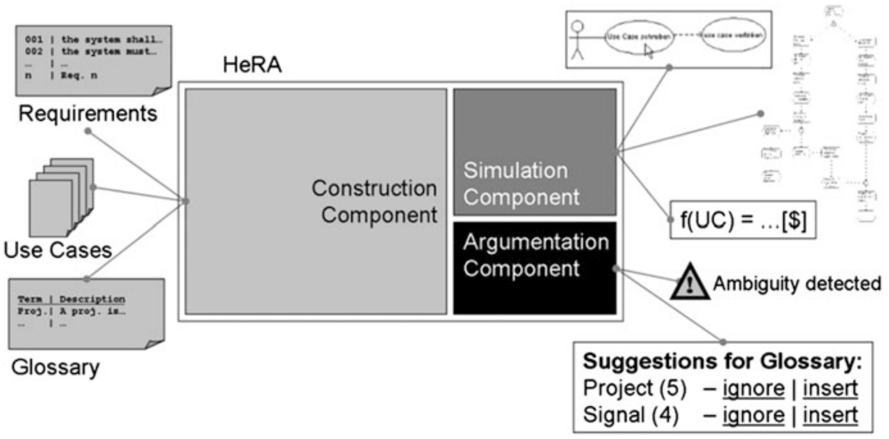


Fig. 15.2 Structure of the HeRA tool

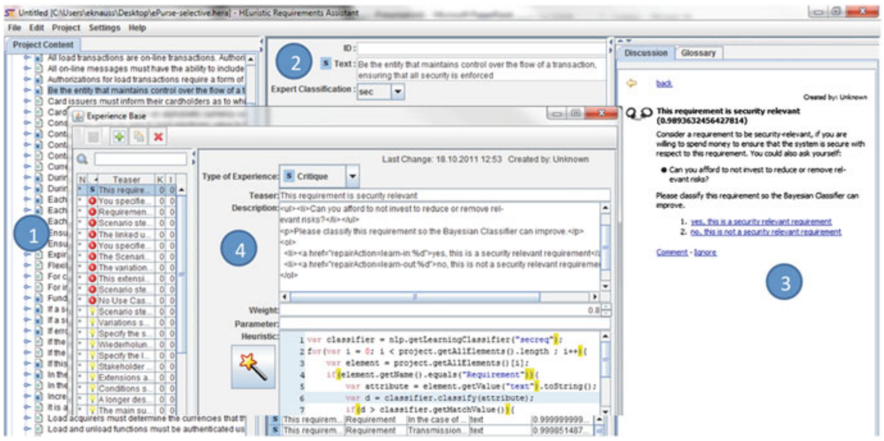
1. Elicitation of user goals on a level of detail that allows for the identification of conflicts
2. Discussion whether the current user goals fit into the underlying business goal and the other user goals already documented (based on visualisation as UML use case and process models)
3. Discussion of important terms and identification of conflicting interpretations of these terms
4. Discussion of prioritisation and project constraints based on the use case points

The direct feedback of HeRA allows starting with elicitation of user goals and detection of inconsistencies and conflicts very early by applying computer-generated feedback.

HeRA's capability of applying direct feedback while writing the document is achieved by automatically evaluating heuristic critiques in the background, whenever a user works on a use case in HeRA. If a heuristic rule fires, the message is displayed besides the use case form and a warning symbol is displayed at the specific field of the use case where the potential problem was detected (see Fig. 15.3, points (1) and (2)).

Additionally, the argumentation component shows a short description for each fired heuristic rule. The user can decide to get a more detailed description for the experience that is encoded through this rule (see Fig. 15.3, point (3)). He can comment on this rule. This feedback can be used to re-evaluate the heuristics and maybe fine-tune them by hand. The user can also decide to ignore this rule if he finds it inappropriate for his current document.

Heuristic critiques can be directly changed in HeRA (see Fig. 15.3, point (4)). This allows rapid prototyping of new critiques. All users can change the message of the critique or parameters (e.g. keyword lists). In addition, the heuristic rule can be adjusted. This rule is encoded in JavaScript. In the scope of the script, all use cases written in HeRA can be accessed.


Untitled [C:\Users\eknauss\Desktop\ePurse-selective.hera] - HEuristic Requirements Assistant
File Edit Project Settings Help
Project Content
Experience Base
Discussion
Glossary
1
2
3
4

Text: Be the entity that maintains control over the flow of a transaction, ensuring that all security is enforced
Expert Classification: sec

Last Change: 18.10.2011 12:53 Created by: Unknown
Type of Experience: Critique
Teaser: This requirement is security relevant
Description:
```
<ul><li>Can you afford to not invest to reduce or remove rel-
evant risks?</li></ul>
<p>Please classify this requirement so the Bayesian Classifier can improve.</p>
<ol>
 <li><a href="repairAction=learn-in:%d">yes, this is a security relevant requirement</l
 <li><a href="repairAction=learn-out:%d">no, this is not a security relevant requireme
</ol>
```
Weight: 0.8
Parameter:
Heuristic:
```
var classifier = nlp.getLearningClassifier("secreq");
for(var i = 0; i < project.getAllElements().length ; i++){
    var element = project.getAllElements()[i];
    if(element.getName().equals("Requirement")){
        var attribute = element.getValue("text").toString();
        var d = classifier.classify(attribute);
        if(d > classifier.getMatchValue()){
```

back
Created by: Unknown

This requirement is security relevant (0.9893632456427814)

Consider a requirement to be security-relevant, if you are willing to spend money to ensure that the system is secure with respect to this requirement. You could also ask yourself:

- Can you afford to not invest to reduce or remove relevant risks?

Please classify this requirement so the Bayesian Classifier can improve.

1. yes, this is a security relevant requirement
2. no, this is not a security relevant requirement

Comment - Ignore

Fig. 15.3 The HeRA critique environment

15.3.1 *Argumentation Component: Glossaries*

Glossaries are widely accepted as a method to define a common ground for communication in software projects. Their goal is to define terms used in the project-specific domain and gain a common understanding. The usefulness of glossaries has often been reported [9, 10].

We often observe that complicated terms are gladly introduced into the glossary of a given project. However, problems arise when common terms are used differently in the context of a project. An example for this is the term *application*. Most developers think they know how the term *application* is defined (according to Wikipedia as *computer software designed to help the user to perform specific tasks*). However, in a given software project, this term was used as "*the task of a student applying for his thesis*".

While this is of course a domain-specific definition of the term at hand, this scenario is especially dangerous, because developers and customers think they have a common understanding about a given term. Customers know the domain-specific definition of this term for their domain, while developers might just know the common definition.

The problem is that neither customers nor developers are aware of these conflicting interpretations of terms. Therefore, they cannot be expected to solve it without help. The only way of solving this hidden ambiguity is to confront both parties with the inconsistency of their assumed meanings.

In the context of experience-based RE tools, there are two possible ways in which the tool can help. Firstly, it can help to build the glossary by suggesting terms to put into it, and it can help maintaining the glossary by giving feedback on its quality. Secondly, it can increase the awareness that there is a glossary and which terms are defined in it. This is important since a reader may not look up these terms because he is not aware of the conflict.

15.3.1.1 Suggesting Terms for a Glossary

To recommend potentially interesting words to the requirements author, we have to define what makes a term interesting. For a glossary tool in the context of experience-based RE tools, we identified two useful heuristics for identifying terms that should be suggested:

1. *Occurrence matters*: A term that is used frequently is probably relevant to the glossary because it can be misused more often.
2. *Experience matters*: If a term was added to a glossary in another project in the same domain, it probably should be defined in the current glossary, too.

There exists empirical evidence that people write important terms more often in order to clarify them or stress their importance [11]. This makes the occurrence heuristic – although simple – an important and strong method to identify terms that are important for a project. These terms are candidates for being added to a glossary.

However, simply counting the occurrence of each word does not lead to satisfying results: A term can be used in different forms over a document. If these terms were not normalised, the algorithm would suggest both spellings separately. For example, the terms project and projects have the same meaning, but if they are compared literally, they do not match. Hence, the algorithm would list each of them with an occurrence count of 1. Therefore, each term has first to be converted into a defined base form. Methods for this are stemming (e.g. Porter stemming [12]) or lemmatisation.

Because SRS are often written in natural language, each term in the glossary should be a correct term according to grammar rules. This requirement makes stemming unusable for getting a base form and leaves us with lemmatisation. We use a modified spellchecking engine together with the OpenOffice.org[1] dictionaries to get base forms of inflected words.

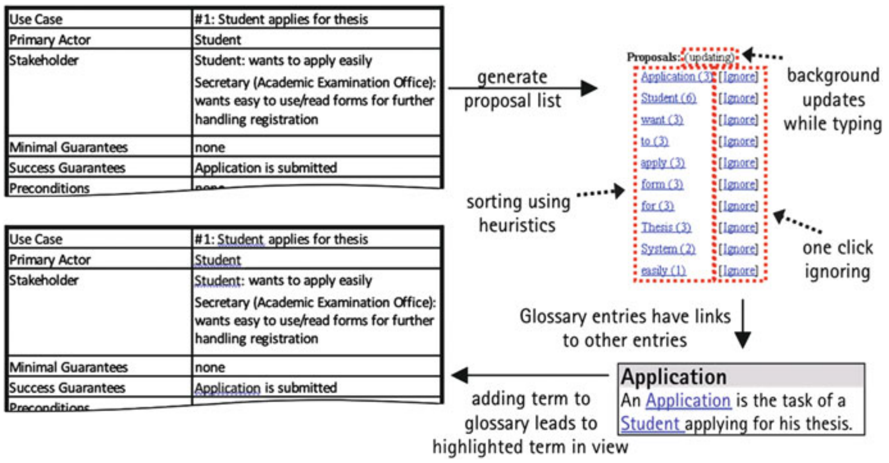
This yields another favourable effect: Since the lemmatisation is language-agnostic, the SRS can be written in every language for which a dictionary exists. We can even have more than one dictionary activated simultaneously. This is, for example, useful for non-English texts containing technical terms in English.

Having solved the problem of different forms of the same term strengthens the occurrence count heuristic. In the above example, project and projects would be suggested as project with an occurrence count of 2.

The second heuristic (*experience matters*) works by remembering the terms, which have already been added to a glossary in another project in the same domain. The experience circle has an “activate” step. This is where tacit knowledge is seeded into the experience circle. The most difficult part is recognising the terms to put into the glossary. Afterwards we can profit from previous experiences and suggest those terms again whenever they are used.

Therefore, a term, which has already been added to a glossary, is not tacit any longer but has successfully been identified to be added to a glossary. Such a term is

[1] http://www.openoffice.org.



Use Case	#1: Student applies for thesis
Primary Actor	Student
Stakeholder	Student: wants to apply easily Secretary (Academic Examination Office): wants easy to use/read forms for further handling registration
Minimal Guarantees	none
Success Guarantees	Application is submitted
Preconditions	none

Use Case	#1: Student applies for thesis
Primary Actor	Student
Stakeholder	Student: wants to apply easily Secretary (Academic Examination Office): wants easy to use/read forms for further handling registration
Minimal Guarantees	none
Success Guarantees	Application is submitted
Preconditions	

Application
An Application is the task of a Student applying for his thesis.

Fig. 15.4 The overall process of working with glossaries in HeRA

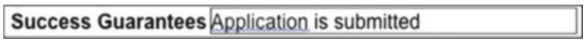
more valuable than a term that is frequently used. Thus, we put those terms on top of the suggestion list, regardless if there are other terms that are used more often, but have not been added to a glossary before.

After identifying the terms that qualify for recommendation, they are shown to the requirements author. For the specification to be helpful, it is necessary to limit the number of simultaneous recommendations. If there are too many recommendations, the user is rather distracted than supported. So we decided to initially show only the ten terms with the highest priority.

Not all words that have been identified as possible suggestions through the two heuristics should be presented to the author. For example, words like *and* or *that* can be filtered out through a stopword list. Furthermore, the author may decide to ignore some words because he does not want them to be part of the glossary, and therefore, they should no longer be suggested. For this case, another filter list is maintained, consisting of all the ignored words. This list is editable by the author. Each author maintains his own list of ignored words. Since they are only plain text files, they can be easily reused or exchanged between different users. Further work may combine creation, maintenance, and exchange of ignored words with social elements, to allow sharing between different users of the same knowledge base. Figure 15.4 shows the overall process of working with glossaries in HeRA.

15.3.1.2 Awareness of Defined Terms

As discussed earlier, the crucial step is to identify ambiguity. This is important both for the requirements author and for anyone who reads the SRS. However, there is one big difference: Since the author has already added the term to the glossary, the former implicit and tacit knowledge about his specific understanding has been explicitly defined. So it is possible to share this special knowledge. That means a



Success Guarantees Application is submitted

Fig. 15.5 Creating awareness for terms in the glossary

reader does not have to bother whether a term is ambiguous or not because he or she can look it up in the glossary.

However, most readers do not look up critical words in the glossary. Either because it stops their natural flow of reading or because they believe a term is not ambiguous. This is the same problem the author faces when writing the document. Consequently the reader has to be supported in identifying ambiguous terms.

We use highlighting to mark terms defined in the glossary. This is a common approach that is, for example, used in almost every modern word processor for spellchecking. This method does not need much space in the text input area but can still easily be recognised by a user. We noticed that we cannot use red as the highlighting colour since this may let the user think of a spelling error.

The use of highlighting makes the recognition of defined terms very easy. The reader is made aware of an existing definition for a term without having to look it up. Therefore, the natural reading flow is not interrupted. This enhances the awareness of the reader since he notices that a word is in the glossary while looking at its context. This would not be given if he looked up the word in the glossary by himself. To further foster the context awareness, tooltips can be used to show the definition of the term. Figure 15.5 shows an underlined term (*Application*) in the HeRA use case form.

15.3.2 More Heuristic Feedback

Glossaries are not the only part of HeRA that takes advantage of the heuristic feedback mechanism. Other examples are Cockburn's aforementioned rules for writing effective use cases that can also be encoded as heuristic critiques.

A more sophisticated example for applying heuristic rules constructively during the creation of a SRS is the SecReq heuristics [13]. These heuristics support the identification of security-relevant requirements. We first created a body of knowledge from three industrial-level specifications. For two of them, we classified the requirements with the help of security experts; for the third, we made use of an already existing requirements database. We then used this body of knowledge to train a Bayesian classifier to recognise security-relevant requirements. The Bayesian classifier was added to HeRA's critique mechanism. This allows HeRA's argumentation component to suggest security-relevant requirements that need further refinement. In addition, the classifier can be trained further by adding information about falsely classified requirements, similar to email spam filters (see Fig. 15.3).

This is an example of a more sophisticated mechanism. In our experience, it is important to have a high learning ability in a tool. This can be achieved by either having very easy mechanisms that can be adjusted by the user (e.g. heuristic JavaScript rules) or by sophisticated algorithms with an easy-to-use interface.

15.4 Overview of Related Tools

Requirements are often specified using natural language, if only as an intermediate solution before formal modelling. As natural language is inherently ambiguous [14], several approaches have been proposed to automatically analyse natural language requirements in order to support requirements engineers in creating good requirements specifications [15–19]. Typically, such approaches define a specific quality model first. Then indicators are defined for the quality aspects that can be automatically evaluated. A good example for this approach is the ARM tool by Wilson et al. [15]. Often these indicators are based on simple mechanisms, for example, keyword lists. Newer approaches leverage sophisticated analysis of natural language, for example, the search for under specification in the QuARS tool [20]. Fabbrini et al. report that the QuARS tool can effectively assess the quality of requirements documentation [20]. Fantechi et al. have applied both the QuARS and the ARMS tool on use cases [21]. They report that use cases are especially well suited for automatic analysis because of their structure. Melchisedech's work goes beyond these approaches by using input from specific workflow and information models for the automatic checks in the ADMIRE tool [22]. This allows for a closer investigation of relationships between requirements but renders the approach only useful if a given workflow is applied.

Somé describes a method and a tool (UCed) to systematically create use cases [23]. The method prescribes a strict use case metamodel and a grammar for natural language descriptions. By constricting natural language to a formalised subset, certain ambiguities and inconsistencies can be avoided.

Jang proposes a formal language for specifying requirements that is founded on mechanisms from knowledge management [24]. The language has similarities to Prolog and allows to easily checking even complex relationships between conditions. Hunter and Nuseibeh propose a related approach that also documents and checks requirements based on logical expressions [25]. They extend the classical logic to allow automatic reasoning on inconsistent requirements descriptions. Such formal and logical languages have advantages in automatic checking but disadvantages in the usability of writing and reading requirements. Thus, Gervasi et al. describe how to transform natural language requirements into a logical representation for verification [26].

Berenbach proposes an approach for automatic checks of UML models, based on heuristics that create an analysis model [27]. This analysis model allows checking, whether UML requirements models have sufficient quality (e.g. for verification or for automatic requirements extraction). The heuristics can be used interactively by the modeller or analytically by the quality assurance. Souza et al. propose to use critique systems to check software engineering models for inconsistencies [28]. Accordingly, such critique systems allow efficient and scalable consistency checking because of the rather small and local critiques.

Kof, Lee et al. work on extracting semantics from natural language texts [16, 17] by focusing on the semiautomatic extraction of an ontology from a requirements document. Their goal is identifying ambiguities in requirements specifications. Gleich et al. present

Table 15.2 Adherence of experience-based requirement tool to design principles

Name	Authority	Proactivity	DEG of interpr.	Learning ability
HeRA	●●●○○	●●●●●	●●●●○	●●●○○
ARM [15]	●●●●○	○○○○○	●●●○○	●●○○○
QuARS [20]	●●●●○	○○○○○	●●●○○	○○○○○
ADMIRE [22]	●●●●○	○○○○○	●●●●○	○○○○○
Chantree [18]	●●○○○	●○○○○	●●●●●	●●●●○
Kiyavitskaya et al. [3]	●○○○○	○○○○○	●●●●●	○○○○○
Jang [24]	●●●●●	○○○○○	●●○○○	○○○○○
Hunter [25]	●●●●●	○○○○○	●●○○○	○○○○○
Gervasi et al. [26]	●●●●●	○○○○○	●●●○○	○○○○○
Berenbach [27]	●●●○○	●●●○○	●●●○○	○○○○○
Souza et al. [28]	●●●○○	●●●○○	●●○○○	●●○○○
UCed [23]	●●●●○	●○○○○	●●●●○	○○○○○

a tool that is able to detect a comprehensive set of ambiguities in natural language requirements [19].

Chantree et al. describe how to detect noxious ambiguities in natural language requirements [18] by using word distribution in requirements to train heuristic classifiers (i.e. how to interpret the conjunctions *and*/*or* in natural language). The reported results (recall $= 0.587$, precision $= 0.71$) are useful in the described context but are too low for more generic approaches as discussed by Kiyavitskaya et al. as discussed in Sect. 15.2.

Table 15.2 gives an assessment of related experience-based requirements tools based on the properties defined in Sect. 15.2. Based on this assessment, related work seems to concentrate on authority and degree of interpretation, thus giving sophisticated and reliable feedback. In contrast, learning ability and proactivity seem to be underrepresented. Because of the impact of these properties on individual and organisational learning, we consider this to be a gap in research that should be closed.

We feel supported in this by Gervasi's discussion on why ambiguity is not always bad [29]. He argues that our language has evolved to cope with uncertainty and missing knowledge. Thus, people are able to articulate this missing knowledge in ways that are then identified as ambiguities. Removing these ambiguities can only be beneficial, if the underlying uncertainty is removed.

Further, Adam et al. propose to approach requirements engineering in a domain-specific way, thus capturing the peculiarities of the domain as soon as possible [30]. The rationale is that for a given domain, the creation of a solution depends on domain-specific information needs. We conclude from this that generic rules have their limitations and experience-based tools, which can be adapted to a specific domain, are a relevant concept.

15.5 Evaluating Experience-Based RE Tools

In Sect. 15.2, we presented concepts for RE tools that go beyond the current state of automatic requirements checking by introducing a learning model. In this section, we discuss strategies for evaluating these experience-based RE tools, based on the research questions we introduced in Sect. 15.2.3. We do this by sketching the evaluation of HeRA, showing that these concepts hold, that the learning model is valid, and that experience-based requirements tools are feasible and beneficial. For details, we refer to other work.

15.5.1 Research Method

For investigating the research questions, we apply the following methods: First, we investigate related work for relevant evidence. Then we find or create a suitable exemplary implementation of an experience-based requirements tool (e.g. HeRA). We let typical users use this exemplary implementation, either in a case study or in an experimental setting. Finally, we complement the observation by using questionnaires to tackle individual learning of these users.

15.5.2 Sketch of Evidence

Here we will give a rough sketch of our evidence that experience-based requirements tools are feasible and beneficial.

Step 1. Existing automatic requirements checkers can be regarded as experience-based requirements tools, because they incorporate experience about typical problems in requirements documentation.

Step 2. Related work (Sect. 3.4) shows that such tools are feasible (as they exist) and beneficial: Some tools are able to cut costs for quality assurance (RQ 3.b); others help to create better requirements documentation (RQ 3.a). We can support RQ 3.a with empirical evidence that students create better requirements documentation with the HeRA (see [31]).

Step 3. We can conclude from this that users improved the documentation based on the heuristic feedback. Thus, they have received (RQ 2.a: reuse), understood (RQ 1.b: reflect), and applied (RQ 1.a: apply) the experience transported by the feedback. During the evaluation reported in [31], we used questionnaires to interview the students. They reported in majority that they have learnt from using HeRA (RQ 1.b: reflect).

Step 4. In an experiment we could show that it is possible to encode new experiences as heuristic critiques in less than 7 min and to change existing heuristic critiques in less than 2 min [32] (RQ 2.b: encode).

15.5.3 Discussion of Implications

In the scope of this chapter, we do not give details about the evaluation. Instead, we show how to approach the research questions raised in this section. Thus, we complement the design principles by giving hints on how to operationalise those principles. In addition, we show that automatic requirements checkers can be even more valuable for an organisation, if they are seen as a way to capture and apply knowledge and experience. Our evaluation shows that it is possible to support the complete learning model: Experience-based requirements tools are improving the documentation of requirements (by lowering documentation cost or increasing quality) and support learning on the organisational and individual level.

15.6 Summary

In this chapter, we argued that experience-based tools can offer important support for requirements engineers. Such tools offer valuable experience during requirements engineering activities. Feedback based on these experiences helps analysts to cope with the information overload and complexity of modern systems.

Beyond recall and precision, we discussed additional properties of such experience-based tools. Especially the proactivity and the ability to learn new experience can be valuable to manage requirements knowledge.

We illustrated our concepts with an exemplary implementation of an experience-based requirements engineering tool. Based on this implementation, we sketched a strategy for evaluating such tools. It is important to evaluate the impact of such tools on the quality of the product (i.e. software requirements specification) and on the quality of the process (i.e. the documentation and quality assurance of requirements). Beyond that, the effect of such tools on organisational and individual learning should be assessed to fully cover the capabilities of such tools. The learning model we presented in this chapter is an important asset for this task. On the long run, improvements of these aspects of knowledge management can lead to considerable benefits.

References

1. Schneider K (2009) Experience and knowledge management in software engineering. Springer, Berlin
2. Senge PM (1993) The fifth discipline: the art and practice of the learning organization. Century Business Random House, London
3. Kiyavitskaya N, Zeni N, Mich L, Berry DM (2008) Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. Requir Eng 13:207–239
4. Schön DA (1983) The reflective practitioner: how professionals think in action. Basic Books, New York

5. Fischer G (1994) Domain-oriented design environments. Autom Sof Eng 1:177–203
6. Knauss E, Lübke D, Meyer S (2009) Feedback-driven requirements engineering: the heuristic requirements assistant. In: Proceedings of the IEEE 31st international conference on software engineering, IEEE, Vancouver, Canada, pp 587–590
7. Cockburn A (2001) Writing effective use cases. Addison-Wesley, Boston
8. Kusumoto S, Matukawa F, Inoue K, Hanabusa S, Maegawa Y (2004) Estimating effort by use case points: method, tool and case study. In: Proceedings of the 10th international symposium on software metrics, IEEE, Chicago, USA, pp 292–299
9. Maciaszek LA (2007) Requirements analysis and system design. Pearson Education Limited, Harlow
10. Berry DM, Kamsties E, Krieger MM (2003) From contract drafting to software specification: linguistic sources of ambiguity, Technical report, University of Waterloo
11. Luhn HP (1958) The automatic creation of literature abstracts. IBM J Res Develop 2:159–165
12. Willett P (2006) The Porter stemming algorithm: then and now. Program Electron Lib Inform Syst 40:219–223
13. Knauss E, Houmb S, Schneider K, Islam S, Jürjens J (2011) Supporting requirements engineers in recognising security issues. In: 17th international working conference on requirements engineering: foundation for software quality, Essen, Germany, pp 4–18
14. Berry DM, Kamsties E (2003) Ambiguity in requirements specification. In: do Leite Prado JCS, Doorn JH (eds) Perspectives of requirements engineering. Kluwer, Norwell, pp 7–44
15. Wilson WM, Rosenberg LH, Hyatt LE (1997) Automated analysis of requirement specifications. In: Proceedings of the 19th international conference on software engineering (ICSE'97. ACM, New York, pp 161–171
16. Kof L (2005) Text analysis for requirements engineering. Ph.D. thesis, Technische Universität München, Germany
17. Lee SW, Muthurajan D, Gandhi RA, Yavagal DS, Ahn G-J (2006) Building decision support problem domain ontology from natural language requirements for software assurance. Int J Softw Eng Knowl Eng 16:851–884
18. Chantree F, Nuseibeh B, de Roeck A, Willis A (2006) Identifying nocuous ambiguities in natural language requirements. In: Proceedings of the 14th IEEE international requirements engineering conference. IEEE Computer Society, Minneapolis, pp 56–65
19. Gleich B, Creighton O, Kof L (2010) Ambiguity detection: towards a tool explaining ambiguity sources. In: Wieringa R, Persson A (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ). Springer, Essen, pp 218–232
20. Fabbrini F, Fusani M, Gnesi S, Lami G (2001) An automatic quality evaluation for natural language requirements. In: Proceedings of the seventh international workshop on RE: foundation for software quality (REFSQ 2001), Interlaken, pp 150–164
21. Fantechi A, Gnesi S, Lami G, Maccari A (2002) Application of linguistic techniques for use case analysis. In: Proceedings of IEEE joint international conference on requirements engineering, Essen, pp 157–164
22. Melchisedech R (2000) Verwaltung und Prüfung natürlichsprachlicher Spezifikationen. Ph.D. thesis, Fakultät Informatik, Universität and Stuttgart, Germany
23. Somé SS (2006) Supporting use case based requirements engineering. Inform Softw Technol 48:43–58
24. Jang H-C (1994) A knowledge-based analyzer for requirements specification analysis. In: Proceedings of the sixth international conference on tools with artificial intelligence, New Orleans, USA, pp 276–282
25. Hunter A, Nuseibeh B (1998) Managing inconsistent specifications: reasoning, analysis, and action. ACM Trans Softw Eng Methodol 7:335–367
26. Gervasi V, Zowghi D (2005) Reasoning about inconsistencies in natural language requirements. ACM Trans Softw Eng Methodol 14:277–330

27. Berenbach B, Borotto G (2006) Metrics for model driven requirements development. In: ICSE'06: Proceedings of the 28th international conference on software engineering. ACM, Shanghai, pp 445–451
28. de Souza CRB, Oliveira HLR, da Rocha CRP, Gonçalves KM, Redmiles DF (2003) Using critiquing systems for inconsistency detection in software engineering models. SEKE, San Francisco, USA, pp 196–203
29. Gervasi V, Zowghi D (2010) On the role of ambiguity in RE. In: Wieringa R, Persson A (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ). Springer, Essen, pp 248–254
30. Adam S, Doerr J, Eisenbarth M, Gross A (2009) Using task-oriented requirements engineering in different domains – experience of application in research and industry. In: Proceedings of the 17th IEEE international requirements engineering conference (RE'09), Atlanta, pp 267–272
31. Knauss E, Flohr T (2007) Managing requirement engineering processes by adapted quality gateways and critique-based RE-tools. In: Proceedings of workshop on measuring requirements for project and product success, Palma de Mallorca, Spain
32. Knauss E, Schneider K (2012) Supporting learning organisations in writing better requirements documents based on heuristic critiques. In: Regnell B, Damian D (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ'12). Springer, Heidelberg/Essen, pp 165–171
33. Knauss E, Schneider K, Stapel K (2009) Learning to write better requirements through heuristic critiques, IEEE, Atlanta, USA