

# Experimenting with Error Abstraction in Requirements Documents

Filippo Lanubile

Dipartimento di Informatica  
University of Bari  
Bari, Italy  
+39 80 544 3261  
lanubile@di.uniba.it

Forrest Shull

Inst. for Adv. Computer Studies  
Computer Science Department  
University of Maryland  
College Park, MD, USA  
+1 301 405 2721  
fshull@cs.umd.edu

Victor R. Basili

Inst. for Adv. Computer Studies  
Computer Science Department  
University of Maryland  
College Park, MD, USA  
+1 301 405 2721  
basili@cs.umd.edu

## Abstract<sup>1</sup>

In previous experiments we showed that the Perspective-Based Reading (PBR) family of defect detection techniques was effective at detecting faults in requirements documents in some contexts. Experiences from these studies indicate that requirements faults are very difficult to define, classify and quantify. In order to address these difficulties, we present an empirical study whose main purpose is to investigate whether defect detection in requirements documents can be improved by focusing on the errors (i.e., underlying human misconceptions) in a document rather than the individual faults that they cause. In the context of a controlled experiment, we assess both benefits and costs of the process of abstracting errors from faults in requirements documents.

## 1. Introduction

Previous experiments [3, 5, 11] have evaluated Perspective-Based Reading (PBR), a method for detecting defects in natural-language requirements documents. In each case, defect detection effectiveness was measured by the percentage of faults<sup>2</sup> found in the requirements

documents reviewed. (The documents used contained seeded faults that were known beforehand.) Major results were that: (1) team detection rates were significantly higher for PBR than for ad hoc review methods, and (2) the detection rates of the individual team members also improved in some instances when PBR was used.

The idea behind Perspective-Based Reading (PBR) is that reviewers of a product (here: requirements) should read a document from a particular point of view. Specifically, we focus on the perspectives of different users of the requirements within the software development process, for example,

- A tester, making sure that the implemented system functions as specified in the requirements;
- A designer, designing the system that is described by the requirements;
- A user, who needs to be assured that the completed system contains the functionality specified.

To support the reader throughout the reading process we have developed operational descriptions (called scenarios) for each role. A scenario consists of a set of activities and associated questions that tells the reader what he should do and how to read the document. As the reader carries out the activities on the requirements, the questions focus his attention on potential defects for his

---

<sup>1</sup> This work was supported by NSF grant CCR9706151 and UMIACS.

<sup>2</sup> We use the following terms in a very specific way in this paper, based on the IEEE standard terminology [8]:

- An *error* is a defect in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. In the context of software requirements specifications, an error is a basic misconception of the actual needs of a user or customer.

- 
- A *fault* is a concrete manifestation of an error within the software. One error may cause several faults and various errors may cause identical faults.
  - A *failure* is a departure of the operational software system behavior from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure.

We will use the term *defect* as a generic term, to refer to an error, fault, or failure.

perspective. Typically, each question is focused on uncovering one type of fault.

In order to create these questions, we used the following taxonomy of faults in natural language requirements. These categories are based on cases in which a necessary attribute (as defined by [1]) of requirements documents have not been met. They are also similar to the taxonomy proposed in [2].

- **Missing Information:** Some significant requirements or software responses are either not included or not defined
- **Ambiguous Information:** A requirement has multiple interpretations due to multiple terms for the same characteristic, or multiple meanings of a term in a particular context
- **Inconsistent Information:** Two or more requirements are in conflict with one another
- **Incorrect Fact:** A requirement asserts a fact that cannot be true under the conditions specified for the system
- **Extraneous Information:** Information is provided that is not needed or used
- **Miscellaneous:** Typically, defects having to do with organization, such as including a requirement in the wrong section

Note that we do not claim these classifications to be orthogonal. That is, a given defect could conceivably fit into more than one category.

Through our experiences in the PBR experiments, we noticed that quantifying, classifying, and defining individual faults was very difficult and possibly subjective. Concentrating on faults had some distinct drawbacks:

- It was often possible to describe the same fault in many different ways.
- Some faults could be associated with completely different parts of the requirements.
- There were groups of similar faults, and it was difficult to understand whether these were best thought of as one or multiple defects.

We began to wonder if measuring defect detection effectiveness by means of the number of faults found was actually a good measure. From a researcher's point of view, focusing on the underlying errors might well be more appropriate. Concentrating on errors rather than individual faults seemed to offer the following potential benefits:

- **Improved communication about the document:** We reasoned that it would be easier for reviewers to discuss and agree on which areas of functionality the document author specified incorrectly or incompletely, rather than on every isolated mistake.
- **Improved learning by developers:** There is a need to go beyond defect removal [9] and focus on improving

the development skills of the participants in a review activity. Looking for basic misconceptions allows reviewers to learn what the actual problems are and then prevent the injection of faults in the future.

- **Domain-specific guidelines for requirements:** Errors for a specific problem domain are expressed at a higher level of abstraction than faults, and could be made available throughout the organization to other reviewers and developers. This benefit is not encouraged by current review methods, which focus merely on defect removal [9].

In order to continue adding knowledge in this area, we have run a new controlled experiment. The primary goal of this study, which is presented here, is to understand whether there is a role for error information within the defect detection/removal process for requirements specification documents. Specifically, we are seeking to evaluate whether error abstraction can be both a *feasible* and *effective* way of augmenting the defect detection process.

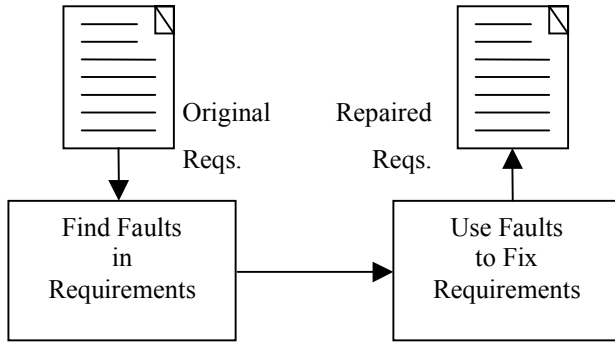
## 2. The Defect Detection/Removal Process

Because the cost of defect removal grows exponentially with respect to the phase in which the defect was injected [4], human reviews, such as Fagan's inspections and its variants [6, 7, 12], focus on finding faults at the output of each phase, by means of individual reading and team meetings.

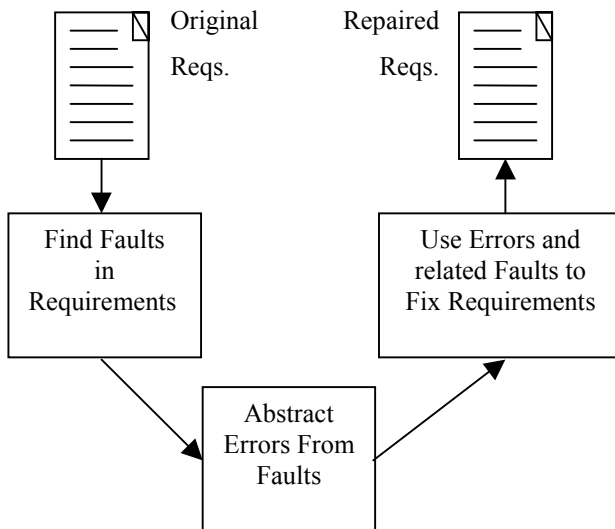
Requirements defects are the most expensive to correct if they are not detected for a long time, and currently formal inspections of requirements documents yield only a low rate of fault detection [10]. For these reasons, we thought it most appropriate to use requirements documents to begin experimenting with the use of errors in defect detection.

Figure 1 shows the usual process to detect and remove defects in a formal technical review. First reviewers find faults in the requirements document, by individual reading and meeting as a team, and then the author uses the reported faults to fix the requirements.

If we find that error information can be useful in the defect detection process, we can augment the usual process to use this additional data. Figure 2 shows our proposed process to detect and remove defects in requirements documents. In the middle of the overall process we introduce a new phase where reviewers are asked to find errors by abstracting from discovered faults. In order to fix the requirements, the author of the document can then receive the recovered list of errors as well as the mapping to the specific faults.



**Figure 1: Usual defect detection/removal process**



**Figure 2. Proposed process with error abstraction**

From our experience as reviewers of requirements documents, we have observed that requirements errors tend to be domain-specific, i.e. errors are more often related to specific functionalities in the system rather than to generic causes that might apply to any system. In the absence of a knowledge base of domain-specific errors, the only chance to discover meaningful errors is starting from what we can observe, i.e., faults in a document, and looking for common patterns at a higher level of abstraction. The error abstraction phase is made up of three main steps:

1. Understand the nature of the faults found, i.e., *why* each fault represents a defect in the document
2. Identify errors by classifying the faults under progressively more abstract themes
3. For each error, make sure to locate all of the faults for which it is responsible

The detailed guidelines for abstracting errors from faults are given in the appendix.

### 3. The Empirical Study

In order to understand the role of error discovery in improving software quality, and to assess our proposed process for reviewing requirements documents, we ran a first empirical study. Our research questions are the following:

- Can this process be used for abstracting faults to errors?
- Does the abstraction process produce meaningful errors?
- What is the right level of abstraction for an error?
- Do subjects get a unique result? (I.e. is there some objective list of errors that subjects can largely agree on, or is the error list largely the result of each subject's point of view?)
- What is the cost of error abstraction?
- What is gained from the process of abstracting errors?
- Can we build reading techniques that effectively encapsulate the new model (reading for fault detection and abstracting errors) for requirements review?

In the following subsections, we present the design of the controlled experiment and provide some early indications from the collected data.

#### 3.1. Experimental Design

In running this experiment in a classroom environment, we were subject to a number of constraints. Most importantly:

- Because all members of the class should be taught something, we could not have a control group. In other words, we could not have a segment of the class with no treatment or training.
- Because subjects were not isolated, and would have presumably discussed their experiences at various parts of the experiment amongst themselves, we could not "cross" the experimental artifacts, that is, use a particular artifact for one group before treatment and for another group after treatment.

We therefore used an experimental design, shown in Table 1, in which the entire group of subjects moves together through six sessions, three of which examine the effect of the error abstraction process followed by team meetings on ad hoc reviews, and a further three which repeat the same examination for PBR reviews.

Thus, we manipulated the following independent variables:

	<b>Individual Fault Detection</b>	<b>Error Abstraction</b>	<b>Team Meeting</b>
<b>Ad Hoc review</b>	Session 1A		
<b>with</b>	↘	Session 1B	
<b>Reqmts Document 1</b>		↘	Session 1C
<b>PBR review</b>	Session 2A		
<b>with</b>	↘	Session 2B	
<b>Reqmts Document 2</b>		↘	Session 2C

**Table 1. Experimental Plan**

- The error abstraction: In sessions 1B and 2B all subjects perform the error abstraction process. Activities performed in sessions 1A and 1B can be treated as a baseline corresponding to the absence of the experimental treatment (i.e., subjects review documents without error abstraction).
- The team meeting: In sessions 1C and 2C subjects meet as teams to discuss error and fault lists. This helps us understand how difficult it would be to integrate the use of error abstraction with inspection meetings. For example, we were unsure of whether teams will report different errors than individuals, because team lists might reflect the consensus of potentially different points of view on errors. Also, since we did not know how similar individual error lists would be, we could not estimate how long teams would take to come to a consensus on an error list.
- The reading technique: Each reviewer used Ad Hoc reading and PBR in sessions 1A and 2A, respectively. The activities of sessions 1B-1C and 2B-2C are also tied to the reading technique because they have as input the fault lists produced as a result of the individual fault detection.
- The requirements document: Two SRSs, one for an Automated Teller Machine (ATM) and the other for a Parking Garage control system (PG), are reviewed by each subject. Both documents contain 29 seeded faults and are, respectively, 17 and 16 pages long.
- The review round: each subjects participates in two complete reviews: session 1 and session 2.

Since each subject is exposed to each level of the independent variables, the experiment has a repeated-measures design. However, the last three factors (reading technique, requirement document, and review round) are completely confounded. That is, we cannot distinguish between the effects of using ad hoc vs. PBR, or reviewing ATM vs. Parking Garage documents. We can also not detect any learning effects that might have made the documents easier to review over time.

We collected initial list of faults discovered (sessions 1A and 2A), the list of abstracted errors and a revised list of faults (sessions 1B and 2B) and the final list of faults and errors that result from team discussion and merging (sessions 1C and 2C).

For the experimental purposes, we measured subjects' response from

- Semi-structured questionnaires (after each session)
- Focused in-class discussion (after session 2C)

The questionnaires asked our subjects to answer multiple questions on key dependent variables in the study:

- Time spent
- Percentage of faults/errors expected to have been found
- Degree of help provided by the abstraction process
- Degree of confidence with the review method taught
- Degree of confidence that the errors produced from the abstraction process actually represent real misunderstandings of the document's author
- Degree of process conformance
- Degree of satisfaction with the reading technique for fault detection
- Degree of satisfaction with the abstraction process
- Variance of individual fault/error lists within a team
- How well the final team fault/error lists corresponded to the original individual lists
- Type of process followed during the team meeting
- Perceived benefits from meeting as a team (if any)
- Perceived differences in abstracting errors between the PBR and ad hoc methods
- Perceived differences in meeting as a team considering that team members had used different perspectives

The in-class discussion was based on the questionnaire answers. They were run to give subjects the chance to clarify, confirm, or expand on their written feedback.

### 3.2. Running the Experiment

The experiment was run as a series of assignments for the CMSC 735 and MSWE 609 classes on Fall 1997. Because the assignments were mandatory and had to be graded, all 65 students participated in the experiment. We assigned each session of the experiment as a homework exercise. However, individual sessions were started in class to allow our students to ask questions about the techniques as needed, and then complete the work outside of class.

At the beginning of the course, we asked students to fill in a questionnaire to assess their experience in areas related to the PBR perspectives: designer, tester, and use case creator. We then assigned subjects to one of the three PBR perspectives in such a way as to provide equal numbers of reviewers at each level of experience. (Note that each subject used only one of the PBR perspectives during the course of the study.) Then, teams were created so that one member came from each of the PBR perspectives and all members of a team had the same level of experience.

Before session 1A, we provided the fault taxonomy to be used as a basis for the ad hoc review, and we gave a lecture in which we provided an example of each of the fault types. The examples came from a requirements document for a video store which was used exclusively for the training lectures. Before session 1B we provided the guidelines for error abstraction and gave a lecture to our subjects. We did not provide any specific training, before session 1C, as to how a team should accomplish the task of discussing and producing a representative list of faults and errors. We therefore allowed the teams to run the meetings however they felt was most useful, and asked subjects to explain the process they used on the questionnaire.

Before session 2A, we gave a lecture in which we presented the PBR technique for each perspective and a short application of the technique to the example requirements document. We also gave handouts containing the detailed perspective-based technique and the associated data collection forms. The training for session 2B was already given in session 1B. Session 2C, like session 1C, did not require training.

### 3.3. Early Results

Here, we present a first analysis of the mix of quantitative and qualitative information, which comes from fault and error lists, debriefing questionnaires and in-class discussion.

#### Can this process be used for abstracting faults to errors?

36% of students indicated that the abstraction process "helped a lot" to identify errors in the requirements, 59% answered that the process provided "some help", and 5% answered that they received "no help". Considering the

cumulative percentage we can say that almost all students found the abstraction process a useful way of composing an error list.

From in-class interviews we discovered that students tended to use (perhaps subconscious) heuristics when they followed the abstraction process:

- Some tried to distribute faults approximately equally among errors
- Some tended to avoid errors that had only a few faults – the tendency was to lump these faults in with other errors
- Some just stopped looking for faults related to a particular error when that error already had a lot of faults compared to others on the list

#### Does the abstraction process produce meaningful errors?

Most students (44/65) had confidence that at least some of the errors they listed represented real, meaningful misunderstandings in the documents.

#### What is the right level of abstraction for an error?

Students reported an average of 9 errors and 33 faults each (for an average of 3.7 faults/error).

However, not knowing whether their abstraction level was "right" was one of the most commonly reported frustrations with the use of error abstraction.

#### Do subjects get a unique result?

At a high level, there were some similarities between the error lists reported.

- 25 different errors were reported in all
- 4 errors were listed by over half of the reviewers
- 3 errors were listed by over a third

#### What is the cost of error abstraction?

On average, students spent about 1.5 hours on abstraction and one hour re-reviewing the document for additional faults.

While for these variables there were no significant differences between the two sessions (ATM document with ad hoc reading and PG document with PBR), the time to get the initial list of faults was higher ( $p=0.0002$ ) reviewing the PG document using PBR (3.75 hours) than for reviewing the ATM document using an ad hoc technique (2.75 hours).

#### What is gained from the process of abstracting errors?

Knowledge of errors did not lead to many new fault discoveries. On average, students found only 1.7 additional true faults as a result of applying the error abstraction process.

Still, most students (49/65) agreed that "the effort spent to abstract the errors was well-spent". Some of the reasons for having found so few new faults were cited:

- Some subjects tended to not put any real effort into finding new faults after abstraction, if they felt that they had found most of the major ones in the first review.
- Some subjects reported that finding errors helped discard false positives, but not find new faults.

Furthermore, fault detection was not felt to be the main point of abstraction. Many other benefits from having performed the abstraction were cited:

- It helped focus on most important areas of the document.
- It gave a better understanding of the real problems in the requirements (63% of students said they were "very confident" that the errors they found really represent true information about the reviewed document).
- It gave more confidence in the faults found.
- It improved communication by being able to talk of general classes of faults
- It seemed to convey better information for correcting the document (although defect correction was not part of the experiment).

Can we build reading techniques which effectively encapsulate the new model (reading for fault detection and abstracting errors) for requirements review?

Interestingly, the three PBR perspectives (designer, tester, and user) seemed to affect the error abstraction process differently. For example, there were some specific comments on the design perspective:

- It seemed to be better at leading to *errors* than *faults* (12 designers agreed vs. 5 disagreed)
- It tended to help find omission faults. (Some subjects felt that these faults were closer to the level of errors in this document. Omission faults often dealt with missing functionality, which indicated that system behavior was not well understood.)

There were also some specific comments on the test perspective:

- The documents were a little too high-level to really use this perspective effectively. Faults could pass the test cases but still be caught because they were clear contradictions to the domain knowledge.
- In contrast to design, this technique tended to catch omissions *within*, but not *of*, requirements.

### 3.4 Limits and Lessons Learned

Here we report other comments from class discussion which point up limits of this study that might be removed in further studies:

- There was some debate as to whether the abstraction process can produce real errors from seeded faults. Since the faults were seeded individually, not starting from seeded errors, it was felt that the results of error abstraction for these documents were necessarily arbitrary. (And if so, mightn't different results be obtained for a document with real faults and errors?) This debate really hinges upon the question of how representative the seeded faults were in the experiment, and we had to admit we really didn't know.
- Our guidelines for the process may have given a piece of bad advice, since they advised that errors that had only 1 or 2 associated faults might not be general enough. It seemed to be agreed that some useful errors could have such small numbers of faults.
- There may have been some confusion of results between faults and errors because some people had already applied some sort of abstraction in the review session. This was especially true in the second session, as having done the whole process once already, subjects reasoned they were just going to have to turn their faults into errors eventually anyway. To some extent, this was a result of the types of faults in the documents: for faults that ended up affecting the document in multiple places it was really just easier to write one error that covered all cases. This was especially true for the faults of omission, because it usually seemed to be large functionalities that were missing and not small pieces.

## 4. Conclusions

In this paper, we have reported our experiences from an experiment designed to assess the usefulness of errors, rather than just faults, in requirements defect detection. The primary indication from this study is that our process of error abstraction appears feasible. Subjects were able to perform the process in what we consider a reasonable amount of time. Although there was some variation in the results among subjects, the process did result in a number of errors that were arrived at independently by a sizable fraction of the subjects. In addition, a majority of the subjects expressed confidence that the errors they produced represented real misunderstandings in the document, and were not arbitrary constructs.

It is more difficult to assess the effectiveness of the error abstraction process for defect detection. The process was certainly not effective in the way we had anticipated: it did not lead to a substantial number of new faults being detected in the document. Our subjects did seem to feel

the process was worthwhile, however, and suggested a number of other ways in which it could be beneficial to the error abstraction process. These alternative measures of effectiveness remain to be tested in future experiments.

## Acknowledgments

Our thanks to the students of CMSC 735 and MSWE 609 classes for their cooperation and hard work.

## References

- [1] ANSI/IEEE. IEEE Guide to Software Requirements Specifications. Standard Std 830-1984, 1984.
- [2] V. Basili, D. Weiss. Evaluation of a software requirements document by analysis of change data. In *Proceedings of the 5<sup>th</sup> International Conference on Software Engineering*, San Diego, CA, March 1981.
- [3] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørungård, and M. Zelkowitz. The empirical investigation of Perspective-Based Reading. *Empirical Software Engineering: An International Journal*, 1 (2): 133-164, 1996.
- [4] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs: NJ, 1981.
- [5] M. Ciolkowski, C. Differding, O. Laitenberger, J. Muench. Empirical Investigation of Perspective-based Reading: A Replicated Experiment, Technical Report ISERN-97-13, April 1997.
- [6] T. Gilb, and D. Graham. *Software Inspections*. Addison-Wesley Publishing Company, Inc., Reading, Mass, 1993.
- [7] W. S. Humphrey. *Managing the Software Process*. Addison-Wesley Publishing Company, Inc., Reading, Mass, 1990.
- [8] IEEE. *Software Engineering Standards*. IEEE Computer Society Press, 1987.
- [9] P. M. Johnson. Reengineering inspection. *Communications of the ACM*, 41 (2): 49-52, February 1998.
- [10] G. M. Schneider, J. Martin, and W. Tsai. An experimental study of fault detection in user requirements documents. *ACM Transactions on Software Engineering and Methodology*, 1 (2): 188-204, April 1992.
- [11] S. Sørungård. Verification of Process Conformance in Empirical Studies of Software Development. Ph.D. thesis, Norwegian University of Science and Technology, 1997.
- [12] D. A. Wheeler, B. Brykczynski, and R. N. Meeson, Jr. *Software Inspection: An Industry Best Practice*. IEEE Computer Society Press, 1996.

## Appendix: Guidelines for Abstracting Errors from Faults

1. Understand *why* each fault you identified represents a defect in the document.

You should first think about the nature of the faults you have identified. Each fault can represent a part of the requirements in which important information was left out, misstated, included in the wrong section, or simply incorrect. Faults might also represent extraneous information that is not important to the system being constructed and may mislead the developers.

Think about the underlying mistakes that might have caused these faults. If information is found in the wrong section of the document, it might mean nothing more than that whoever wrote the requirements organized the document badly. On the other hand, if there is a contradiction within the document, then the requirements may reflect an underlying confusion about some of the functionality that is to be provided by the system. The distinction is not always so clear: if a necessary test is misstated in the requirements, does this represent an isolated typo or a fundamental misunderstanding about error conditions?

For example, consider these faults, which were found in the requirements for the computer system of a video store. They all come from the part of the requirements that details how tape rentals are to be entered into the system.

- F3: The requirements say “The system keeps a rental transaction record for each customer giving out information and currently rented tapes for each customer.” However, an explanation of exactly what information is given out for each customer has been omitted.
- F9: The requirements say that when a tape is rented, the “rental transaction file is updated.” However, what it means to update the rental transaction file is not specified. The information to be stored here is not discussed.
- F10: The requirements say that when the rental is successfully completed, a form will be printed for the customer to sign. They also specify that the forms “will be kept on file in the store for one month after the tapes are returned.” This is classified as a fault because the filing and storage of the forms is outside the scope of the system, and should not be specified in the requirements since no corresponding system functionality will be implemented.

- Identify errors by classifying the faults under progressively more abstract themes.

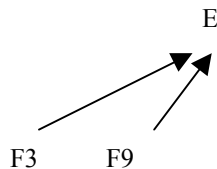
Identifying fundamental errors means finding patterns in a collection of isolated faults. As such, it is a creative process for which we can give only a few guidelines.

It may be helpful to think about what kind of information is involved in the faults. Do particular functionalities, system users, or performance requirements appear in multiple faults? This may indicate that these are particular aspects of the system that are not well understood.

It may also be helpful to remember that not every fault is indicative of a larger error; some will undoubtedly result just from typos or misunderstandings of very small pieces of the functionality. Not every fault has to be a part of a pattern.

Two of the example faults discussed above can help demonstrate this. F3 and F9 both involve missing information about how the information in the database is to be updated. This may be evidence of an underlying error that goes beyond these specific faults: how rentals are to be logged is not completely specified. Faults F3 and F9 may be assumed to appear in the document because this underlying error has been made. F10 may simply represent nothing more important than an isolated fault, but if we could find other faults involving the rental forms, then F10 might be evidence that the role of the rental forms in the system was not completely understood. In that case, we would have to ask whether there isn't an even larger error, namely, that what historical information is to be recorded, and how it is to be kept, are not adequately specified.

We can illustrate F3 and F9 abstracting to a common error in the following way:



Naturally, this may not be the only mapping of faults to errors. Multiple ways of categorizing the faults may certainly exist.

- For each error, return to the document and make sure you have located all of the faults for which it is

responsible.

Finally, return to the document to see if the errors you identified lead you to other faults that you might have missed in the first analysis. Now that you know the fundamental misunderstandings, you can see if those errors showed up in other places as well.

In our example, once we realize that there may be a fundamental error involving the understanding of how transactions are to be logged, we can return to the document and look more closely for other faults that may have resulted. In this way, we could find:

F12: The requirements state that when a customer pays a late fine, the rental transaction record is updated. Again, we realize that the exact fields and values that are to be updated are never specified.

We can think of the new error as leading to other faults in the same category:

