Developing Techniques for Using Software Documents: A Series of Empirical Studies

Ph.D. Thesis Proposal

Forrest J. Shull
University of Maryland

Abstract

This proposal presents an empirical method for developing "reading techniques" that give effective, procedural guidance to software practitioners. Each reading technique is tailored so that it can be used to accomplish a specific software-related task (e.g. defect detection) using a specific kind of software document (e.g. requirements). This empirical method can be followed to create and then continually improve reading techniques, since it explicitly builds and tests underlying models of how developers use a particular software document in a particular environment. That is, our approach for developing reading techniques checks the underlying assumptions about:

- Specific tasks for which the document will be used
- Aspects of the document that must be understood to support those tasks
- Aspects of the environment affecting tasks using the document

This empirical approach avoids the mistake of applying assumptions that are true in other environments to a context in which they have not been tested.

We illustrate the development method by discussing how it was used to create two reading techniques (one for finding defects in requirements, and a second for reusing design and code from an Object-Oriented framework in a new system). We describe a series of studies by which we and other researchers have assessed the effectiveness of these techniques in practical situations. These studies show that these reading techniques are useful and effective guidelines for accomplishing real software engineering tasks, and thus demonstrate the effectiveness of the empirical development method.

Finally, this proposal discusses how indications from these studies have been used to begin to formulate higher-level hypotheses about reading techniques. We discuss the results from these earlier studies as well as important questions that have not yet been addressed. We propose a further study that is designed to test potential improvements to one of our reading techniques and, in the process, shed some light on the unanswered questions from our earlier studies.

Table of Contents

1 Introduction

This proposal presents a body of research that makes two main contributions to software engineering:

1. It presents an iterative process for creating and improving reading techniques, which are procedural techniques that give effective guidance for accomplishing a specific software-related task. We refer to this process simply as the Reading Technique Development (RTD) process. As will be explained in greater detail in the rest of this chapter, reading techniques created by the RTD process provide an effective way to use a given software document (such as requirements) for a particular task (for example, defect detection).
2. It presents two reading techniques that have been tailored to different documents and tasks. Aside from providing a demonstration of the effectiveness of the RTD process, these techniques also represent useful and effective procedures that can aid in accomplishing some common tasks in software engineering.

This proposal also discusses a series of studies that provide an empirical evaluation of particular reading techniques, and uses the experimental results to begin building a body of knowledge about this type of technique and their application. As part of this discussion, we demonstrate that our understanding of the experimental methods and results can be packaged in a way that allows others to learn from and extend our findings.

This proposal is organized as follows: Section 1.1 motivates why the RTD process is useful and necessary, by discussing the specific problem it addresses. Section 1.2 presents the RTD process as our proposed solution, provides the relevant definitions, and defines the scope of this work. Section 1.3 outlines how we intend to provide an empirical validation of our proposed solution.

The remaining chapters of this proposal are concerned with studying the effects of our proposed solution in practice. Chapter 2 provides an overview of work we have already undertaken in this area, identifying common themes and explaining the most important open research questions. Finally, chapter 3 proposes a new study to continue this work.

1.1 PROBLEM STATEMENT

Software practitioners are interested in using effective software development processes, and for this they have many tools and processes from which to choose. The field of software engineering has produced many such options, ranging from large sweeping process changes (e.g. moving to an Object-Oriented programming paradigm) to detailed refinements of a particular aspect of the software lifecycle (e.g. using a formal notation rather than natural language to record system requirements). In trying to quantify this phenomenon, some sources have counted that, since the 1970s, "literally hundreds" of different work practices have appeared and have been claimed to somehow improve software development [Law93]. Relying on standards bodies to identify effective processes does not greatly simplify the situation, since over 250 "standard practices" have been published in software engineering [Fenton94].

The dilemma, of course, is how the practitioner is to know which tools and processes to invest in, when almost all of them come with claims of enhanced productivity. All too often, the wrong decision is made and both software practitioners and researchers invest time and money in tools and processes that never see significant use. Tools and processes can fail to find use for several reasons:

- they are too difficult to integrate into the standard way of doing things for an organization;
- they do not pay adequate attention to the needs of the user;
- they require the developer to invest too much effort for too little gain;
- they incorporate techniques or technologies that are not effective in the particular environment;
- the limits of their effectiveness are simply not understood.

Actually, all of these reasons are inter-related. The common theme is that the tools and processes in question simply do not match the requirements of the environment in which they are to be used. Often, even the developers of an approach do not know how effective it will be in a given environment, what its limitations are, or to what classes of users it is best suited.

This situation could be avoided if process development and tools selection were based on an empirical characterization of what works and what does not work in a particular environment. Such an empirical foundation avoids ad-hoc selection of software tools and techniques based upon assumptions (about the tasks developers need to do, about the types of information developers need to understand, about characteristics of the product being developed) that are never checked. In their paper, Science and Substance: A Challenge to Software Engineers, Fenton, Pfleeger, and Glass dwell on the plethora of tools, standards, and processes available before pointing out that "much of what we believe about which approaches are best is based on anecdotes, gut feelings, expert opinions, and flawed research." What is needed, they conclude, is a careful, empirical approach [Fenton94].

Basili described the basics of such an empirical approach in his paper, The Experimental Paradigm in Software Engineering. To develop an understanding of the effectiveness of software processes in a particular environment, he stresses the necessity of starting from a real understanding of the software development process, not from assumptions. He advocates:

- modeling the important components of software development in the environment (e.g. processes, resources, defects);
- integrating these models to characterize the development process;
- evaluating and analyzing these models through experimentation;
- continuing to refine and tailor these models to the environment. [Basili93]

As demonstrated in this chapter, we use these guidelines as the basis of our empirical approach.

1.2 PROPOSED SOLUTION

1.2.1 Definitions

This dissertation proposal presents an empirical approach for developing tailored techniques for use by software practitioners. We refer to these techniques as "reading techniques." The word "reading" was chosen to emphasize the similarities with the mental processes we use when attempting to understand any meaningful text; a reading technique is a process a developer can follow in attempting to understand any textual software work document. It is important to note that, while "reading" in software engineering is often assumed to apply only to code, in fact any document used in the software lifecycle has to be read effectively. (For example, requirements documents have to be read effectively in order to find defects in inspections.) We address the problem space in more detail in section 1.2.2.

More specifically, a reading technique can be characterized as a series of steps for the individual analysis of a textual software product to achieve the understanding needed for a particular task [Basili96b]. This definition has three important components:

1. *A series of steps:* The technique must give guidance to the practitioner. The level of guidance must vary depending on the level best suited to the goal of the task, and may vary from a specific step-by-step procedure to a set of questions on which to focus.
2. *Individual analysis:* Reading techniques are concerned with the comprehension process that occurs within an individual. Although the method in which this technique is embedded may require practitioners to work together after they have reviewed a document (e.g. to discuss potential defects found individually in a document), the comprehension of some aspect of the document is still an individual task.
3. *The understanding needed for a particular task:* Reading techniques have a particular goal; they aim at producing a certain level of understanding of some (or all) aspects of a document. Thus, although we may think about CASE tools to support a technique as it evolves, the focus of this research is not on the tools themselves but on the developer's understanding of artifacts (whether supported by tools or not).

The research presented in this proposal demonstrates a proof of concept that our approach to developing software reading techniques can be effective, by providing empirical evidence concerning the effectiveness of the specific techniques developed. We also propose to validate our approach by testing whether it can be used to build up knowledge about reading techniques in general. These goals are broad but we narrow them to more detailed evaluation questions in section 1.3.

To be as clear as possible about the scope of the work presented in this proposal, we use the following definitions:

- Technique: A series of steps, at some level of detail, that can be followed in sequence to complete a particular task. Unless we qualify the phrase, when we speak of "techniques" in this proposal we are referring specifically to "reading techniques," that is, techniques that meet the three criteria presented at the beginning of this section.
- Method: A management-level description of when and how to apply techniques, which explains not only how to apply a technique, but also under what conditions the technique's application is appropriate. (This definition is taken from [Basili96].)
- Software document: Any textual artifact that is used in the software development process. This definition encompasses artifacts from any stage of the software lifecycle (from requirements elicitation through maintenance), either produced during the development process (e.g. a design plan) or constructed elsewhere but incorporated into the software lifecycle (e.g. a code library).

Our definitions sharply differentiate a technique from a method. Perspective-Based Reading (PBR), discussed in section 2.1, provides a useful example of each. PBR recommends that reviewers use one of three distinct *techniques* in reviewing requirements documents to find defects. Each of these techniques is expressed as a procedure that reviewers can follow, step by step, to achieve the desired result. PBR itself is a *method*, because it contains information about the context in which the techniques can be effectively applied, and about the manner in which the individual techniques should be used in order to achieve the most thorough review of the document.

1.2.2 Focusing on Software Documents

One of the most important characteristics of our approach to developing reading techniques is the focus on providing guidance for how developers use software documents. Software documents are important because they contain much of the relevant information for software development and are used to support almost all of the developer's tasks. Developers are required every day not only to construct work documents associated with software development (e.g., requirements, design, code, and test plans) but also to analyze them for a variety of reasons. For example:

- requirements must be checked to see that they represent the goals of the customer,
- code that was written by another developer must be understood so that maintenance can be undertaken,
- test plans must be checked for completeness,
- all documents must be checked for correctness.

In short, software documents often require continual understanding, review, and modification throughout the development life cycle. The individual analysis of software documents is thus the core activity in many software engineering tasks: verification and validation, maintenance, evolution, and reuse. By focusing on software documents we focus on the sources of information that are crucial for the entire development process.

For example, our primary research environment in the NASA Software Engineering Laboratory (SEL) uses a model of software development that consists of eight different phases (requirements definition, requirements analysis, preliminary design, detailed design, implementation, system testing, acceptance testing, maintenance & operation). Each phase is characterized by specific tasks and the products that they produce [SEL92]. From observation of this environment we draw two important conclusions about software documents:

- They contain valuable information about the software under development.
- They are an important way in which information is passed from one phase to another.

By focusing on these documents, we focus on one of the most important information sources about the development of a system. Although other software lifecycles may have more or fewer than eight stages, and may define the stages themselves differently, work documents are always created and used as repositories of information about the development process. This research concentrates on how developers can best be supported in understanding these documents for all types of tasks.

The effective use of software documents requires two types of tasks from developers [Basili96b]:

1. **Analysis Tasks** are aimed at assessing various qualities and characteristics of a document. Techniques that support them are important for product quality, as they can help us understand the type of defects we cause, and the nature and structure of the product. Analysis tasks are aimed at answering questions such as:
   - Does a document satisfy some property or criterion (e.g. information in the document can be traced to, or is consistent with, another document)?
   - Does the document possess a particular attribute (e.g. correctness, consistency)?
   - Does the document contain any defects (according to a particular definition, e.g. the document contains ambiguities that might prevent a user from implementing the final system correctly)?

2. **Construction Tasks** are aimed at using an existing system artifact as part of a new system artifact. Thus construction tasks include reusing a document from a previous version of a system to document the current system, using a document from an earlier activity (e.g. a requirements document) in the construction of a later document (e.g. a design document), or incorporating a document developed outside the current environment in a system. Construction techniques are important for comprehending what a system does. They are also important for understanding what components a document (or the system it describes) contains, how these components relate to each other, and how these components can be integrated with the system being built. These tasks are aimed at answering questions similar to the following:
   - Does the document (or system it describes) contain information that can be useful to the new system? (For example: Does a code repository contain functions that would be useful to the

system under construction? Does a user manual contain use cases that could also be used to describe the functionality of the new system?)

- If a component from the document (or the system it describes) is used in the new system, what other components must also be used? Is it possible to determine what parts of the system will be affected? (For example, supposing a function is used from the code repository, what other functions in the repository does it reference that must also be reused? How can it be determined whether the reused functions have any side effects that might affect other functions already implemented?)

Different types of tasks can be applied to the same document. For instance, a requirements document could be created in the requirements phase and analyzed for defects (an analysis task) before the system moves to the design phase. That same requirements document could later be used as a source of reuse in a later project, in which the system to be built is expected to have similar functionality. The requirements document for this system could be created by reusing some descriptions of system functionality from the previous system (a construction task), rather than writing the new document completely from scratch.

Although different models of the software lifecycle may affect how such tasks are undertaken, all environments will need to apply some kind of tasks to software documents. A waterfall model may require an analysis task to be undertaken at the end of every phase of the lifecycle, to catch as many defects as possible in the documents produced before they are used in future stages. In contrast, a rapid prototyping model of development may iterate through the phases of the lifecycle multiple times, relying on the analysis of the current prototype in each iteration to identify defects from previous phases (which are then fixed on the next iteration through the lifecycle). In this environment, construction tasks may be more appropriate: they may be used to understand how documents from previous versions of the system may be used to construct the next version.

In short, software documents and the tasks in which they are used are important and integral parts of software development. We feel that developers can benefit from formal guidance for these tasks because, although developers are often taught how to *write* code and other artifacts, the required skills for effective *reading* of existing artifacts are seldom (if ever) taught and must be built up over time with experience [Basili97]. This omission has been noticed especially with program code, the artifact upon which most computer science education is focused, and calls for teaching and assessing code reading skills appear periodically in the literature (e.g. [Koltun83], [Deimel90]). Some basic research has in fact been focused on developing code reading techniques, such as Mills' reading by step-wise abstraction [Linger79]. "Reading" skills do, however, apply to other software documents, if by "reading" we refer to the process of visual inspection for understanding.

For the most part, however, these activities have been largely ignored both in research and practice. For example, methods such as walk-throughs, reviews, and inspections (applied to various documents in the lifecycle) attempt to improve the detection of defects in software documents, not by changing the ways in which developers understand the documents, but by changing what they do with this understanding (e.g. explain their understanding to other developers, look for particular characteristics based on their understanding). At the level of the individual reviewer, however, none of these methods provide any guidelines for how the reviewer can examine the document effectively in search of defects [Knight93]. Put another way, such methods simply assume that the given document can be reviewed effectively while ignoring the actual individual processes that are taking place [Basili96].

Although every reading technique is tailored to one specific task involving software documents, each document can have many associated tasks and hence many associated reading techniques. This can lead to a bewilderingly large problem space, as there are a large number of tasks (and variations upon them) undertaken in different software development environments. Worse yet, any given type of document can appear in any of a variety of formats, styles, and notations. In order to understand the set of reading techniques and be able to reason about the similarities and differences between them, we have developed a taxonomy (see Figure 1) of the set of possible reading techniques, based on the document and task to which the technique has been tailored [Basili96b]. Our motivation in this undertaking is the assumption that

reading techniques that accomplish similar tasks or are applied to similar documents have other similar features as well.

Each level of the taxonomy represents a further specialization of the problem according to the attributes that are shown in the rightmost column. At the grossest level, we organize all possible tasks into either Construction or Analysis tasks, as defined earlier in this section. Next, we group tasks according to the specific objective they aim to achieve. These specific tasks are suggested by the product qualities that are to be achieved in the final software. For example, to achieve correctness in the system we need to undertake fault detection; to achieve reliability we need to check traceability, among other things; and to achieve efficiency we need to assess performance. Some of these tasks may be applied in different forms to documents at different phases of the lifecycle; e.g., we may undertake fault detection in both requirements documents and code, which would require very different techniques. We can also think of some tasks that apply to documents from multiple phases of the lifecycle; e.g., an Object-Oriented framework is an artifact that is advertised to support reuse of both design and code. Finally, the last level of the taxonomy addresses the very specific notation in which the document is written. For example, requirements can be written in a natural language, or translated into a formal notation for requirements such as SCR. In section 2.1.1 we discuss briefly an example of how switching notation (but leaving all other variables constant) can affect the reading technique used.

We can use this taxonomy to describe the techniques for Perspective-Based Reading (which focuses on defect detection in English-language requirements, and is described in section 2.1). We say the PBR techniques are tailored to analysis (high-level goal), specifically to detect faults (specific goal) in a requirements specification (document) written in English (notation/form). [Basili96b]

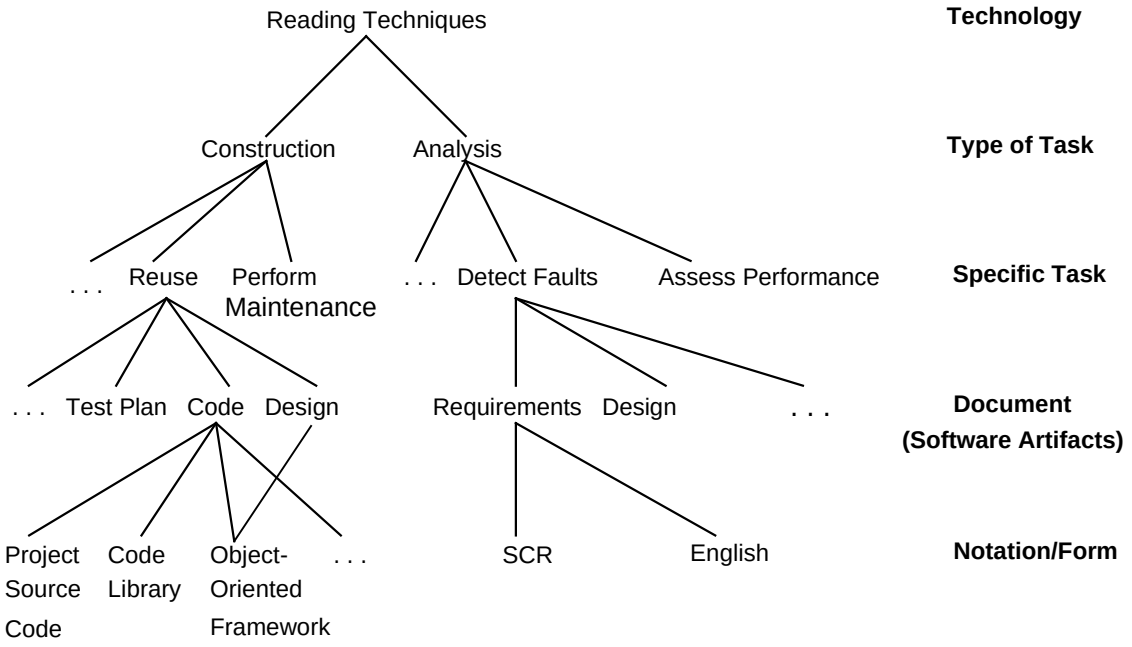

Figure 1: Partial problem space addressed by software reading techniques.

This taxonomy can be used to help specify more exactly the problem to be addressed by a particular technique. As we continue this research and construct more techniques, we hope to use the taxonomy to begin to develop broader, more general theories about software reading techniques. For example, it could be applied to answer research questions such as: Are there characteristic differences between techniques that support analysis and those that support construction? Can particular activities be reused for any technique addressing a specific task?

Currently, however, these questions remain unanswered. Because of the wide variation among the attributes of software documents used in different environments (e.g. notations, formats, levels of detail) we cannot start out by formulating an over-arching theory of the best way for developers to use a particular type of document for a development task. The techniques we develop must therefore rely on an empirical approach to understand what works and what doesn't work in practice in a particular environment. Although we begin to address larger questions in this work by carefully choosing the reading techniques we study, and packaging our work and results in a way that assists its extension by other researchers, definitive answers to these questions are outside the range of this proposal.

1.2.3 An Empirical Approach

We argue that an empirical basis is necessary for an approach to developing software techniques. Since we lack detailed models that allow us to reason about software processes and products in general - and more importantly, since we often lack knowledge about the limits of software methods and techniques in different contexts - any reading technique which is created must be evaluated under realistic conditions before we can make any assumptions about its effectiveness. Methods that appear to solve a general problem may not be applicable in many specific instances, and an empirical basis avoids the mistake of applying assumptions which are true in other environments to a context in which they have not been tested.

Empirical approaches to software process improvement are not new. (Examples are the Shewart-Deming Plan-Do-Check-Act Cycle and the Quality Improvement Paradigm, both of which have been used in industry to improve software development processes [Basili95].) Such empirical approaches are related to the scientific method where one observes the world, establishes a hypothesis, collects data, and then analyzes the data to determine whether or not the hypothesis is supported. In software development, this approach usually takes the form of hypothesizing that a given change to the development process will have a beneficial effect (e.g. lowering costs, reducing effort, improving final quality), making the change, and then seeing if the result in practice supports the hypothesis or not.

Our Reading Technique Development (RTD) process, based on the scientific method, can be summarized as:

- Observe the way the document is used in the environment. The results of this observation are encapsulated in two types of models: models of what information in the document should be important for the given task (the **abstractions of information**) and a model of the process by which the task is achieved (the **uses of information**). A reading technique for accomplishing the task is generated by using each model to suggest a step-by-step procedure. These procedures are, respectively, the **abstraction procedure,** which assists the user in understanding the abstractions, and the **use procedure**, which provides guidance for using the abstractions in the task at hand. Each of these procedures accomplishes a part of the task, and when combined into the finished technique the entire task is addressed. (We discuss this process in more detail in the next section, but introduce the terms here to help clarify the discussion.)
- Formulate a hypothesis concerning this technique. If the technique is newly created, we hypothesize that the technique represents an improvement over the current way the task is accomplished; if a previous version of the technique is in use, we hypothesize that the new version represents a significant improvement. (In this proposal we present examples of both cases. Chapter 2 describes the generation and assessment of two new techniques. Chapter 3 proposes further work that generates specific improvements to an existing technique and provides a comparative assessment of the two versions.)
- Collect and analyze data about the procedure. Through executing the reading technique and carefully assessing the results, we can either refute our hypotheses (and conclude that, under the current conditions, the reading technique is not effective) or build up confidence in its effectiveness. It must be noted that it is necessary to resist the temptation to generalize about the worth of the technique based on a single study. Rather, the aim must be to build up a body of significant, reliable evidence that allows us to understand the limits of the reading technique being tested.
- Iterate. The RTD process is meant to provide further improvement for reading techniques based on experiences learned over time. If the execution of the reading technique shows that it is not effective, then the technique itself and the conditions under which it was executed should be observed closely to determine the problem, which can then be addressed by a new reading technique. Of course, new hypotheses should be formulated about this reading technique and analyzed in turn. Even a successful

execution should lead to further observation to identify potential improvements, which can then be assessed in turn.

These bullets outline the RTD process, a mechanism for generating and assessing reading techniques. The process is empirical because the techniques are evaluated in the context in which they are intended to be used, allowing us to build an understanding of what does and does not work in that context. We present RTD in full detail in the next section.

1.2.4 A Process for Developing Reading Techniques

The RTD process is based directly on the Quality Improvement Paradigm (QIP) [Basili85, Basili88, Basili93b, Basili95]. The QIP is an empirical method for software process improvement that has been used effectively in industry [Basili94, Basili95b]. The QIP was selected as the basis for our work on reading techniques because it is explicitly designed for the development of processes, like reading techniques, in which:

- The process will rarely be applied exactly the same way twice (since a reading technique, in normal use, will not be applied multiple times to exactly the same document).
- The time frame does not permit substantial data to be built up for building predictive models (since the application of a reading technique is a non-trivial process that may only need to be performed only once for a software project).
- Variations are often introduced due to the human-centered nature of the process (a reading technique is intended to be used by many different software developers, each with different backgrounds and experiences).

The QIP itself addresses these concerns through a continual "corporate learning" cycle of model-building and testing that is composed of six steps (summarized in Figure 2):

1. Characterize the project and its environment.
2. Set quantifiable goals for successful project performance and improvement.
3. Choose the appropriate process models, supporting methods, and tools for the project.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze the data to provide real-time feedback for corrective action. (This step contains its own, smaller, iterative cycle for learning while the project is being executed. This iteration allows the process to change during execution in order to incorporate feedback, and then continue to be executed and analyzed in real time.)
5. Analyze the data to evaluate current practices, determine problems, record findings, and make recommendations for future process improvements.
6. Package the experience in the form of updated and refined models, and save the knowledge gained from this and earlier projects in an experience base for future projects.

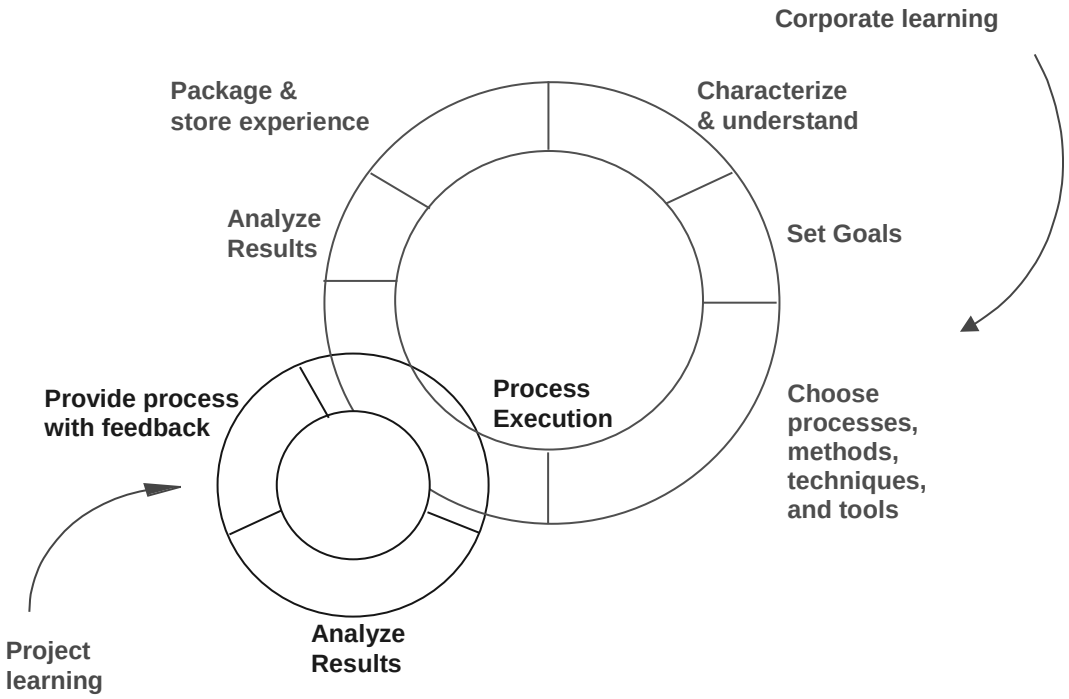

Figure 2: The QIP cycle for iterative process improvement.

The QIP can be adapted for use in developing reading techniques for a particular document. What is necessary is a slightly different focus for each of the main steps, in order to perform activities which are specific to the development of reading techniques. These activities are in addition to, but do not replace, the normal activities associated with process improvement in the QIP. Below, we describe the six steps of the RTD process and discuss how each is a specialized form of a step of the QIP. In the next section, we illustrate the RTD process by following an example through all six steps.

1. Modeling the underlying concepts involved in using the document.

The first step of the QIP is concerned with understanding the software development environment by means of whatever is available (e.g. current lifecycle and process models and data). This step is certainly necessary for reading technique development, but must be especially focused on how the document is used in the environment. At a high level, modeling the document may involve specifying the notation that is to be used, its format, or its content. Modeling the environment may require models of key independent variables, such as:

- the entire software lifecycle, in order to determine in which other stages the document will be used;
- the process used for relevant stages of the lifecycle, and the actual work practices used by developers in this stage, which may have to be obtained through an ethnographic study;
- important characteristics (such as experience) to identify different classes of developers who use the document.

Doing this effectively means being able to recognize which are the right variables from which to create models. Since we are modeling the way in which a document is used to accomplish a task, we focus on developing two types of models: **abstractions of the information** contained in the document, and the **uses of the information** that accomplish the given task. The "abstractions" tell us what information in the document is useful, while the "uses" tell us how the task is completed using that information. Together, these two types of models capture the characteristics of interest.

How to construct these models varies slightly depending on whether the model is to be used for an analysis or construction task. For an analysis task, in which we assess the quality of a document, we have to make sure that the document can support all of the other processes in which it is needed. To do this, we have found questions such as the following helpful:

- In what other phases of the software lifecycle is this document needed?
- What other specific software tasks does the document support, in those phases?
- What information does the document have to contain, to support those tasks?
- What is a good way of organizing that information, so that attributes of interest are readily apparent to the user?

This line of questioning creates a way of thinking about the document that is useful to support the actual processes in which the document will be used. The model of uses can then be created by answering the following type of questions, for each of the tasks identified above:

- How is the document used to support this task? What do these developers do with the information from the document, once they have it?
- Is the information of sufficient quality that it can be used in this way?

Of course, definitions of key terms (such as "sufficient quality") will have to be tailored to the particular task being discussed.

Models for construction tasks are created in an analogous way. We have found a slightly different set of questions helpful here, due to the fact that in a construction task we typically have to support only one process in which the document is used (the construction task itself), rather than consider all possible downstream processes.

- What information does the document contain that is suitable to the new system?
- Can the information be organized into components (e.g. use cases, classes, functions) that can typically be reused as a whole?
- What are my criteria for deciding if a particular component would be useful to the new system? Can I organize the components according to the attributes of interest?

Again, the identification of an organizing principle (here, the components) suggests a model of the uses:

- Can I create a decision process to answer whether or not a particular component could be useful to the new system?
- How do I incorporate useful components into the new system once I've decided they might be useful?

Developing the right abstractions is especially crucial because, in both cases, the abstraction model directly influences the creation of the uses model. Developing abstractions can also be particularly difficult since often, in order to undertake a task effectively, it may be necessary to look at a document in many different ways. We may therefore need multiple abstractions in order to capture all of the relevant information. Finding the right abstractions, and making sure that the set of abstractions provides coverage of all the important information, are both important considerations.

2. Setting the evaluation criteria

In the second step of the QIP we set quantifiable goals for what it means to be a "successful" project in this environment. This step is concerned with choosing criteria on which the reading technique can be evaluated, so that we can understand if it has a positive impact in our environment. Setting the criteria correctly obviously depends on choosing measures that are capable of accurately evaluating the effectiveness of a technique. The characterization of the environment in the previous step must be kept in mind to ensure that our success criteria are both relevant and reasonable.

Multiple success criteria can and should be chosen to evaluate whether the technique satisfies all important factors. It should also be considered that valid criteria can range from the subjective (e.g. an increase in faults found per hour, a decrease in the mean time to failure of the

finished system) to the objective (e.g. the level of satisfaction with the technique must be rated "high" by a majority of the users).

Although the specific evaluation criteria must be based on the task and environment, we have also identified broad evaluation questions that may be useful for any reading technique. We discuss these questions in section 1.3.1, which contains a more specific discussion of the entire validation strategy used in this research.

3. Mapping the model into a concrete process.

This step is crucial for reading technique development. In most applications of the QIP, this step usually asks that we choose an appropriate process to use for accomplishing our task, based on the characterization and goal-setting already done. For reading techniques, we use this step to translate the models that were developed in the characterization step (step 1) into guidance for practitioners.

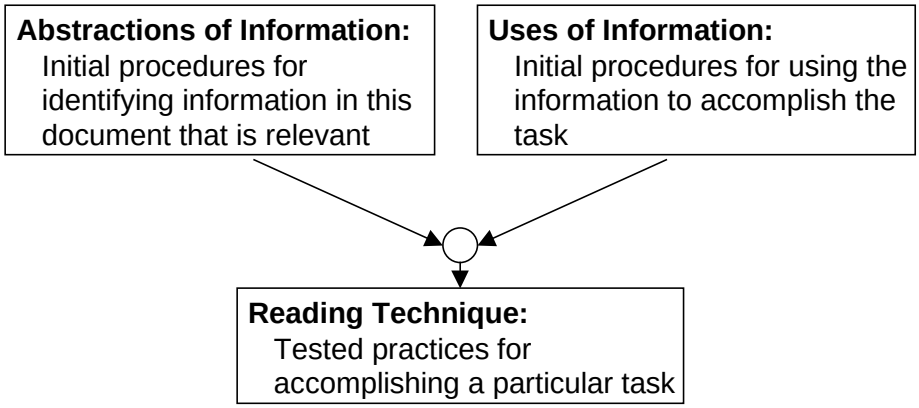

Figure 3: Building focused, tailored software reading techniques

As we build the reading technique, it must be based closely on the abstraction and use models in order to ensure that the technique is tailored to the particular document and work environment (Figure 3). We have found the following process helpful in this regard:

1. The abstractions of information in the document are used to create an initial **abstraction procedure** for identifying the relevant information. This procedure can be thought of as a "traversal" algorithm that instructs the practitioner in how to find the necessary information in the given document. The information can then be analyzed or used in system construction (though this part of the process is handled by the use procedure, below). Creating the abstraction procedure usually involves determining the organizing principles of the abstraction, and how they can be found in the document. It is necessary to focus on relatively independent parts of the abstraction so that practitioners can concentrate on useful pieces of information rather than the document as a whole. This procedure may need to be repeated at different levels of detail, since different features may be important depending on whether the abstraction is being examined at the most general or most detailed level.
2. The uses of information expand the procedure developed in the previous step. In other words, for each of the steps of the abstraction procedure generated above, we need to provide a **use procedure** that contains guidance as to how the information seen so far contributes to solving the task. Creating such a use procedure is especially necessary when the abstraction procedure guides the practitioner to traverse the abstraction at successive levels of detail; different levels of detail may contribute to the solution of the task in different ways.

The abstraction procedure gives the practitioner the right information; the use procedure should provide a step-by-step way to use that information to complete the task.

In putting these two procedures together to develop the technique, it will be helpful to reuse prior knowledge whenever possible. That is, we should borrow from techniques, or incorporate

tools, that have already been proven successful in the environment. In some cases, ethnographic studies may be necessary to understand exactly what techniques and tools developers have already found to be useful. This attention to tested tools can result in a more effective technique for the environment, and aid in overcoming developer resistance to being asked to use a "new" technique.

4. Executing the Technique.

In this step the reading technique is executed, with relevant performance variables being monitored at all times in order to be able to make changes during execution, if necessary. The technique may be executed "off-line," as part of a study to explore its feasibility and effectiveness, or used during a real software development project, where the results can be compared against results for similar tasks and techniques in this environment. In our view, off-line experimentation has a valuable role in reading technique development, as new techniques can often be evaluated and debugged more cheaply off-line than if used immediately on an industrial software development project. This chance to debug the technique off-line lowers the risk and expense associated with a new technique, by identifying problems before they have the chance to cause delays and frustrations on projects facing real deadlines [Burgess95].

As the technique is executed and data about its use are collected, it is necessary to keep in mind the concept of process conformance. How certain can we be that the practitioners are actually using the technique as we intended it to be used? A lack of process conformance is a real danger in that it may lead to a misinterpretation of the results, but it should not be viewed as a sign of malice on the part of the practitioners. It may be that practitioners simply feel uncomfortable applying the new technique, preferring to fall back on more familiar, although perhaps less effective, techniques. It is also possible that the practitioners have not received enough training, and think they are applying the technique when actually their understanding of it is different from what was intended. Regardless, it may be necessary to examine intermediate artifacts that are produced during the execution of the technique, or at least to solicit practitioners' reaction to the technique via questionnaires, informal interviews, or some other method.

Some thought must also be given as to the manner in which this technique is to be tested, in order to get an accurate picture of its effectiveness. There are a host of concerns which, although they cannot always be avoided, should be kept in mind along with their impact on results. These concerns can be grouped into roughly three categories, based upon the way they influence the results [Judd91]:

- Construct validity aims to assure that the study correctly measures the concepts of interest. It requires that we have some confidence that the technique that is being executed is actually the reading technique that we intend to study. (It is a real danger that practitioners may ignore the technique and fall back on other approaches with which they are more familiar.) If we are comparing a reading technique to some other technique, it is of course necessary to make sure that the comparison technique is really a viable alternative and not a "straw man" that is used to make the reading technique look better than it is.
- Internal validity aims to establish correct causal relationships between variables as distinguished from spurious relationships. It is necessary to be sure that the measures we collect and the criteria of effectiveness we have established are really measuring what we think they are. At the same time, we must not overlook any factors that could provide an important explanation for the effects we notice.
- External validity aims to assure that the findings of the study can be generalized beyond the immediate study. External validity is especially important in "off-line" studies, since external validity asks us to question whether the environment, subjects, document, and task with which the technique is tested are really representative of the situation in which the technique will actually be applied if adopted. If not, it is difficult to say what practical knowledge we have gained from executing and studying the technique.

We can afford to give these concepts only a cursory mention here; interested readers are referred to [Judd91] for a more complete discussion. [Basili86, Rombach93, Kitchenham95] are also useful introductions to these concerns in the realm of software engineering research.

This step also represents an important opportunity to collect qualitative data on the application of the technique. (For example: Were the users satisfied? Was it unpleasant to have to follow the technique? What kinds of difficulties were encountered?) All of this data can later be used to validate or improve the underlying models and technique.

5. Analyzing the performance of the technique.

Once the execution is completed, the performance of the technique must be analyzed to determine whether or not it was actually effective. Performing the analysis correctly involves some knowledge of statistical procedures, which is too extensive a body of knowledge to usefully summarize here. As before, we mention only a few important concerns:

- Are we applying proper statistical tests, given the way in which the variables of interest are measured? Different operations can be performed on data, depending on the underlying mathematical scale (e.g. ratio, ordinal, nominal). In measuring software attributes, it is not unusual that an attribute's scale type is not known beforehand. It is thus important that scale types be confirmed for attributes, and that this knowledge then be used to make sure that proper statistical tests are applied.
- Are we applying proper statistical tests, given the distribution of the data sets? Most commonly-used statistical tests make assumptions about the distribution of the population from which the data set is drawn. If these assumptions are not met, any statistical test can give invalid and misleading answers. For any statistical test, it is important to make sure that the underlying assumptions hold.
- Has statistical power been considered? It is all too common that software engineering research is concerned with statistical significance without considering the power of the test or what this says about the size of the effect being measured. An effective analysis should take all of these factors into account when discussing results.

Interested readers are referred to [Fenton97] and [Campbell63] for a more complete discussion and pointers to other relevant literature.

It is important to note that all relevant variables should be accounted for in the analysis in order to get an accurate picture of the effectiveness of the technique. (Effectiveness, of course, must be defined in terms of the criteria set in step 2.) Identifying the relevant variables is especially important for assessing the needs of the intended users of the technique. For example, if multiple classes of users are identified (say, experienced and inexperienced) is the technique equally effective for all classes?

If the analysis shows that the technique was effective, according to our criteria from step 2, then we can take a closer look at the technique to determine if there are any likely areas for improvement. If it is not effective, then we still need to look more closely at the technique in order to determine where the problems were. Interviews or other methods of obtaining feedback from the users of the technique may be invaluable in this regard. The first question must be about the accuracy of our evaluation process. If the technique was used and there are no other confounding factors, then the problem is with the technique itself. We have to determine if the models developed in step 1 were correct, if we selected the right criteria for evaluating effectiveness in step 2, and if the mapping from models to a systematic technique was done well in step 3.

6. Packaging our experiences with the technique.

In this step of the QIP, we consolidate what we have learned about the technique during our latest pass through the corporate learning cycle – that is, during the latest execution of steps 1 through 6 of the QIP. This normally includes a host of practical challenges, such as:

- disseminating this experience to relevant recipients within the organization,
- incorporating any resulting improvements into the larger processes,
- updating the relevant documentation.

The concept that this execution of the RTD process is only our "latest" pass through the cycle is a key one, since the lessons we have learned should form the basis for improvements to the technique that would in turn be evaluated through another iteration of the QIP cycle.

Our approach to packaging experience has been to develop WWW-based "lab packages" which contain the following information:

- Motivation for this study (i.e. what features from our step 1 characterization of the environment are addressed by the technique or by the latest improvements);
- Detailed description of the technique (i.e. a description of our results from step 3, in sufficient detail that it can be critiqued by other researchers or practitioners, or even borrowed for use in different environments);
- Design of the study (i.e. a description of the environment in which the technique was executed, as selected in step 4, and the variables we monitored to determine effectiveness, as chosen in step 2);
- Results of the study (i.e. details of the analysis done in step 5 and the conclusions that we can draw from it).

Such lab packages can support future work on this subject not only by the same researchers but also by independent researchers as well. By providing easy access to our experimental materials, we hope to encourage such independent replications. Although the contributions of replicated studies are too often minimized, such studies would allow the following beneficial effects:

- A strengthening of the *external validity* of the study: By replicating studies in other environments, it becomes less likely that the conclusions of previous studies are the results of characteristics of the experimental environment or subjects rather than the software engineering aspect being evaluated. Replicated studies help us understand the limits of our results.
- A strengthening of the *internal validity* of the study: A replicated study which allows certain variables to vary from previous experiments in carefully controlled ways provides further evidence of which aspects of the experiment interact with the software engineering aspect being evaluated. Such replicated studies give us more confidence in the experimental design.

1.2.5 An Example

In this section we trace the development of a hypothetical reading technique through the six steps of the RTD process. Suppose that we are trying to develop a technique for reusing code from an Object-Oriented class library. We are interested primarily in what functionality is contained in the library; we expect that developers will understand what functionality is needed for the system they are building, and want to examine the library to see if it contains functionality that they can use.

1. Modeling the underlying concepts involved in using the document.

For the abstraction model, we therefore want to develop an abstraction of the library that is organized according to functionality (the attribute of primary interest). We first observe the current practices in our environment, and learn that developers have used several libraries previously, most of which were organized into several hierarchies of related classes. Therefore, we may use this idea of the class hierarchy as the organizing principle for our new reading technique, since it succeeds in organizing the class library by the attribute of interest (functionality). While there are certainly other organizing approaches that we might use instead, the fact that developers in our environment have some previous experience with this one may be sufficient to recommend its use. (Of course, it is necessary to understand what the tradeoffs are in using this model instead of some other. Are we giving anything up in exchange for the opportunity to make our developers more comfortable with the technique? For the purposes of keeping this example short we will stay with the set of hierarchies.)

Our model of the uses of this information is a relatively simple one:

- Compare the functionality in the set of hierarchies against the functionality needed for the new system. If a match is found, then:

- Find the relevant code for the functionality in the library, and
- Integrate the code into the new system (modifying it if necessary).

2. Setting the evaluation criteria

To measure the effectiveness of a technique for reuse from an OO class library, there are many attributes we would want to focus on. For example:

- For successful searches, the amount of time required on average to find a reusable component;
- For unsuccessful searches, the average amount of time required to realize that functionality being sought does not exist in the library;
- How often users have unsuccessful searches when there is in fact functionality in the library that could have been used in the system;
- How satisfied users are with the technique for finding functionality as a whole.

For the purposes of this example, let us concentrate on just two of these in more detail. We can use these attributes to specify evaluation criteria that provide a definition of effectiveness. For example, to be considered effective any technique we develop should:

- *Not* increase the amount of time spent looking for reusable code and
- *Not* decrease the amount of code that is reused.

That is, an effective new technique must be at least as effective with respect to these attributes as the currently accepted way of accomplishing the task.

3. Mapping the model into a concrete process.

When we selected a set of hierarchies as the abstraction model of the OO class library, we implicitly decided that useful entities for organizing the information would be first, the hierarchies, and second, the classes themselves. Thus, when we create the abstraction procedure to provide guidance to the practitioner on how to "traverse" the abstraction, we will need to worry about several levels of detail. The first thing to do is to understand the type of functionality provided by each of the class hierarchies. For each hierarchy, the most important entity is the root class, which must be examined in order to understand the base functionality that will be inherited by every other class in this hierarchy. The next step is to examine the children of the root in order to understand the main sub-types of the functionality provided. Then the children of each of these must be examined in order to understand successively more specific categories of functionality. When we have reached the "leaves" of this hierarchy tree, we have reached the most specific implementations of some class of functionality. If even more detail is necessary, we may choose to go further and actually examine the attributes and methods within the class in order to understand how a very specific piece of functionality is achieved.

Then it is necessary to create the use procedure. The use model suggests a procedure for evaluating useful functionality, which must be tailored to the steps of the abstraction procedure. The first step of the abstraction procedure is to examine the separate hierarchies, so we need to provide a version of the use procedure which determines whether each hierarchy is likely to yield reusable code for the current system; hierarchies that are not likely to contain such functionality need not be searched. The next step of the abstraction procedure involves looking at successive levels of classes within the hierarchy. For this, we need to develop a procedure for determining whether any of the classes at different levels of the hierarchy would be useful for the system. Again, classes that our procedure determines are not likely to be useful can be pruned from the traversal along with all of their children. At the lowest level of the abstraction procedure, we need to supply a version of the use procedure that can be followed to determine whether individual attributes and methods within a class would be useful or extraneous to the current system.

In constructing the procedures, we might talk to the developers and ask them what they have found helpful in addressing this problem in the past. Do they resort to simple code reading? If yes, what are the things that they look for? Are tools available which have been found helpful? This information can be used, where possible, to help the techniques take advantage of what is already in place in the environment and to make use of skills the developers have already practiced.

4. Executing the Technique.

Before asking developers to use the technique that we developed on a real software project, we might run an off-line study to assess its effectiveness first. We might take some volunteers from among the developers and first ask them to demonstrate their current reuse techniques on a practice program using an OO library. Then we train them in the new reading technique, and give them another small example program to write. (In order to be a useful basis for comparison, the practice programs would have to match as closely as possible in terms of their most important attributes, such as the difficulty of implementation and developer familiarity with the problem domain.) As the subjects apply the reading technique on the practice programs, we monitor the amount of time it takes them to turn in the finished implementation, ask them to estimate how much time they spend looking for reusable components each time they use the library, and ask them to indicate which classes and methods come from the library and which were built from scratch. Just as importantly, we must also be available to answer questions if there are problems, such as instructions in the technique that the subjects find confusing, or tools that the technique recommends that the subjects cannot access. All of these problems are recorded and used to improve the technique.

5. Analyzing the performance of the technique.

Let us suppose that after analyzing the results we collected from our off-line study of the reading technique for reuse, we notice that the subjects took about the same amount of time on both of the practice programs. That is, the amount of time required didn't change much regardless of whether our reading technique or the subjects' usual technique was used. At first glance the data may seem to show that the amount of reuse was about the same for both techniques as well. But, if we divide the subjects into two groups based on experience, we may see that experienced developers actually achieved less reuse using the new technique, while inexperienced ones achieved more. If we have confidence in our experimental design and analysis, then we can conclude that the new technique actually was more effective, at least for less experienced users.

6. Packaging our experiences with the technique.

Making the techniques themselves, and the artifacts and results of the experiment, available for other researchers or practitioners to use is relatively straightforward. The more important question is, what do we do with our results? On the basis of one study, it seems premature (to say the least) to insist that all developers in our environment use the new technique. Further study is more prudent, but it is important to design future studies to answer our specific questions. For example, our next step might be to interest a single development team on a real project in the technique, so that we could monitor its use in great detail on a real project. We also need to decide what to make of the indication that the technique is more effective for novices than experienced users. Can it be improved to make it more helpful for experienced developers? We might observe the difficulties they had with it more closely, and try to correct these in another version of the technique, which should then be empirically assessed again. Or perhaps we should conclude that the technique is most useful as a mechanism for training developers who are new to this task, and try to assess it in that context by means of additional studies.

This example demonstrates how the RTD process might be used. Chapter 2 discusses two full examples of its actual use to develop reading techniques that address real software engineering problems. Chapter 3 proposes a third study that illustrates a second iteration through the process for an existing reading technique. That is, we use an existing reading technique as the baseline and execute the RTD process in order to determine necessary improvements.

1.3 VALIDATION PLAN

Validating the proposed solution presented in this chapter requires an evaluation at two levels. First, the reading techniques themselves must be evaluated, as called for by the RTD process. This evaluation is a necessary part of RTD that allows us to measure how well we are meeting the requirements of the environment as we execute the RTD process. Each technique should be evaluated according to criteria that make sense for the environment and task to which it has been tailored, as described in section 1.2.4. However, given the nature of reading techniques, there are also some general evaluation questions that should be addressed for any technique. We discuss these questions in section 1.3.1.

Secondly, the RTD process itself must be evaluated as a potential solution to the problem identified in section 1.1. This is a larger evaluation that must be undertaken to understand if this research is making a useful contribution to the area it addresses. We describe our validation strategy for RTD in section 1.3.2. We use the evaluation questions in section 1.3.1 to sketch a validation plan that aims to assess the RTD as a means of developing reading techniques.

1.3.1 Evaluating Reading Techniques

The RTD process for developing techniques generates many questions. At the highest level are questions of correctness: What are the correct models for a given environment? Have we produced a procedure correctly, based on those models? Have we correctly evaluated the effectiveness of the technique? In the absence of any *a priori* theories about the correct way of creating such techniques, however, we cannot examine each of these questions in isolation. From an empirical viewpoint, the best we can do is to evaluate whether or not the techniques produced are in fact effective in the context for which they were designed. If the answer is yes, then we have some evidence that our models, procedures, and evaluation strategy are effective, at least for a particular type of task in a particular type of environment. If not, then we at least know that there is a problem that we will have to understand and correct in order to apply the RTD process effectively for this task in the given environment.

As discussed in section 1.2.4, an important part of this process is the evaluation of the effectiveness of the reading technique in context. Also, we are interested in two other factors – the experience of the subjects and the level of specificity of the model – which in our experience have proven to be important. We present our evaluation questions below and sketch a plan of attack for each.

EQ1. Is the reading technique effective for the intended task in the environment?

This question proceeds from the part of the definition (section 1.2.1) that says software reading techniques are undertaken *for a particular task*. Obviously, then, the first measure of a successful technique is whether it was effective for the intended task. A plethora of methods can be used to measure this, ranging in formality from surveys to case studies to formal experiments. Where possible, we proceed mainly through quantitative, controlled experiments. Kitchenham *et al.* [Kitchenham95] recommend this approach for evaluating the effectiveness of techniques for stand-alone tasks (i.e. tasks that can be isolated from the rest of the software development process, a description that applies to many of the tasks which lend themselves to reading techniques). They argue that such tasks can be investigated using controlled studies without resorting to "toy" or unrealistic problems, the results can be judged relatively quickly, and small benefits can be more easily discerned since the results can be interpreted in isolation from other processes. If the technique is found not to be effective, then an analysis must be undertaken to answer if this is because of problems either with the models that characterize the document and tasks or with the mapping to a technique.

EQ2. Is the reading technique at the right level of specificity?

This question arises from our definition of a technique as a *series of steps*. This definition implies that when the technique is created, a design decision has to be made about the level of granularity the guidelines should reflect. The correctness of this decision will need to be validated mainly by qualitative analysis, by assessing difficulties and satisfaction with the technique. (A quantitative analysis can be undertaken only when we can compare two techniques that are equivalent in every way except for the level of detail, which would be hard although not impossible to do.)

EQ3. Do the effects of the reading technique vary with the experience of the subjects? If so, what are the relevant measures of experience?

Since a software reading technique addresses *individual analysis* of software documents, it is necessary to see if the effectiveness varies with the attributes of the user, or if "one size fits all." This analysis will have to contain both quantitative and qualitative components. Qualitative analysis is necessary for finding explanations for the observed effects of the technique from the background information on the subjects. Quantitative analysis is necessary for testing for correlations between different measures of experience and the effectiveness with which the technique is applied.

EQ4. Can the reading technique be tailored to be effective in other environments?

This question is necessary because it must be recognized that the work contained in this proposal can be only a starting point for an investigation into building software reading techniques. By studying the effects of the technique in other environments, we can begin to build up an understanding of the limits of the technology. Obviously, much of this effort will depend on other researchers and practitioners who continue to extend this work, but we can make a significant contribution by providing lab packages. Successful lab packages would encourage and aid replicated studies, contributing to the validity of a specific study.

1.3.2 Evaluating the Process for Developing Reading Techniques

The evaluation questions in the previous section are concerned with evaluating a particular reading technique. In this section we discuss the larger question of how to validate, not the reading techniques themselves, but our approach to creating them. We sketch a validation plan and discuss how the evaluation questions for specific techniques can form the basis for a validation of the technique creation process.

The primary criterion for a successful technique development process is whether or not it can produce techniques that are effective in practice. But we can achieve a better level of granularity by breaking the process into its component pieces, and validating each of these in turn. The resulting questions are high-level and inter-related. However, they not only indicate whether the process as a whole is working as expected, but can help identify particular aspects in need of improvement.

The first major component of the RTD process is the actual development of the reading technique, comprising steps 1 and 3 of the process. It would not be useful to consider a lower level of granularity, since we are interested in the underlying models (created in step 1) only insofar as they are useful for creating procedures for the reading technique (in step 3). By considering both steps together we can formulate the first process evaluation question:

PEQ1 Is the RTD process for developing reading techniques feasible and effective?

Given the nature of reading technique development, we can break this down into two more specific questions:

PEQ1.1 Is RTD a feasible way to create effective reading techniques?

PEQ1.2 Is RTD a feasible way to improve reading techniques?

Recall from section 1.2.4 that the QIP, on which RTD is based, claims that its cycle of creating and testing models is useful not only for creating processes from scratch but also for continuously identifying and executing improvements. These questions seek to test whether this holds true for RTD in the domain of reading techniques.

Our second process evaluation question addresses experimentation (steps 2, 4, and 5 of the RTD process):

PEQ2 Does the RTD process build up sound knowledge about important factors affecting reading techniques?

In answering this question we must address a number of specific concerns. For example, the stipulation that the knowledge contributed must be *sound* implies that we must validate whether appropriate and correct experimental designs and analyses are being used to evaluate the reading technique under study. That is, we must assess whether RTD is capable of producing accurate answers to EQ1 (whether a given technique is effective in the environment), introduced in the previous section.

The dimensions along which experimental validity must be assessed were introduced briefly in section 1.2.4. To assess **construct validity** requires that we have some confidence that the technique actually being executed in the experiment matches our assumptions about which technique is being used and at which level of detail. This particular evaluation is necessary to determine how accurate our answers can be for EQ2 (whether a given technique is at a useful level of specificity). That is, we need to know at which level of specificity a technique is being applied before we can draw conclusions about that level of specificity. Assessing **internal validity** requires some confidence that the phenomena we are measuring in the real world do in fact measure "effectiveness." This can be addressed in step 2 of the RTD, in which criteria for effectiveness must be determined to be useful and feasible given the models of the environment and task. Finally, checking **external validity** requires having confidence that the results we see in our studies can be generalized to the real environment in which the technique should be applied. We can gain this confidence in part by having the experiment replicated in other environments, by other researchers. Such replications help demonstrate that the results we observe are not due to features particular to our environment.

Another specific concern addressed by PEQ2 is suggested by the stipulation that studies "build up knowledge about important factors." If the RTD process is well-specified and consistent, then we should be able to learn about factors that influence the creation and effectiveness of reading techniques, as we apply the process to different tasks in different environments. Thus, we must validate whether the use of RTD leads to studies that do not only address a single reading technique, but can be used in conjunction with other studies in order to begin to build up knowledge about the set of all reading techniques. In this way, a series of studies might be used to elicit hypotheses about the effect on reading techniques of factors such as the type of task or document to which the technique is tailored. If RTD studies cannot be combined in this way, then questions such as EQ3 cannot be answered, since we could not begin to draw general hypotheses about results noticed in multiple studies.

It should be noted that an answer to PEQ2 is a necessary prerequisite to addressing process evaluation questions PEQ1.1 and PEQ1.2. That is, we cannot say whether the use of RTD leads to "effective" or "improved" reading techniques without having validated the process for assessing effectiveness and improvement.

Finally, we can validate step six of the RTD process by asking:

PEQ3 Can lessons learned from the RTD process be packaged for use in other environments?

The last step of RTD calls for the results of the latest iteration of modeling, testing, and analyzing a particular reading technique to be packaged so that it can serve as the baseline for the next iteration. However, as we discussed in the last section, in order to build a fundamental understanding of reading techniques we must evaluate them in many different environments. Thus our packaging must be sufficiently broad that other researchers are able to tailor any given technique themselves, and continue future iterations in their own environments. It is necessary to assess whether such packaging can be created in order to determine whether RTD can provide an answer to EQ4 (whether reading technique can be tailored to other environments).

2 Completed Work

We have created two families of reading techniques for using documents at different points of the software lifecycle:

- Perspective-Based Reading, for defect detection in requirements documents
- Scope-Based Reading, for reuse of design and code in Object-Oriented frameworks

We have run experiments to evaluate the practical effectiveness of these techniques. In each case, we have learned much that can be applied to the process of modeling, mapping, and evaluating such techniques.

In this chapter, we discuss our work on these previous experiments. We describe each experiment by following the outline of the RTD process. We begin by describing the abstraction and use models we created for the specific task, and how they are mapped to procedures and used to create a reading technique (RTD steps one through three). We sketch the experimental design and analysis (steps four and five) to assist the reader in judging the strength of our conclusions, and provide references to more complete descriptions of the experimental details. Finally, we discuss how each experiment addresses the four evaluation questions presented in section 1.3.1:

EQ1. Is the reading technique effective for the intended task in the environment?
EQ2. Is the reading technique at the right level of specificity?
EQ3. Do the effects of the reading technique vary with the experience of the subjects? If so, what are the relevant measures of experience?
EQ4. Can the reading technique be tailored to be effective in other environments?

As part of the answer to EQ4 we discuss how the experiment and its results were packaged for use in other environments (step 6).

In section 2.3 we discuss what the lessons learned from these experiments contribute to the validation of the RTD process.

2.1 INVESTIGATING A TECHNIQUE FOR ANALYSIS OF REQUIREMENTS: PERSPECTIVE-BASED READING[1]

2.1.1 Introduction

Our first investigation was into a technique for an Analysis Task, the first main branch of our taxonomy tree (recall Figure 1). We decided to focus on defect detection in requirements documents, because of an earlier study [Porter95] which had shown the effectiveness of a family of techniques called Defect-Based Reading (DBR) for this task. DBR was defined for analyzing a requirements document written in SCR-style notation [Heninger85], a formal notation for event-driven process control systems. Because a formal notation was assumed for the requirements, each technique in the DBR family could be designed to focus on particular defect classes (e.g., missing functionality and data type inconsistencies) which were associated with the notation. In that study, DBR was compared against ad hoc review and checklist-based review to evaluate its effect on defect detection rates. Major results of interest were that:

- It was shown that review effectiveness could be increased by providing reviewers with detailed procedural techniques, since DBR reviewers performed better than ad hoc and checklist reviewers with an improvement of about 35%.
- It was shown that perspectives could be used to focus reading techniques on different aspects of the document being analyzed in an effective, orthogonal manner. Reviewers using each of the different DBR techniques did better at finding the type of faults their technique was designed to detect, but did about the same as any of the other techniques at finding other defects.

So, DBR had shown that defect detection in requirements was an area that could benefit from the introduction of tailored reading techniques. We decided to focus on this area for further experimentation.

Our study was designed to complement the study of DBR in order to gain a deeper understanding of the effectiveness of reading techniques in requirement fault detection. The crucial difference, however, was in choosing English-language requirements, to see if the conclusions about DBR would also apply to documents that were much less structured and written in a less specific notation. This implied that the underlying models of the technique would have to change: we abandoned the idea of separate perspectives based on defect classes (which had been chosen for reasons specific to the SCR notation) and developed new models.

2.1.2 Related Work

Most of the current work in reviewing requirements for defects owes a large debt to the very influential works of [Fagan86] and [Gilb93]. In both, the emphasis is on what we would term the review *method* (using the definition presented in section 1.2.1). They concentrate on what tasks should take place before the inspection meeting, how the meeting should be organized, and what should be the goal of the meeting itself. They do not, however, give any guidelines to the reviewer as to how defects should be found; both assume that the individual review of these documents can already be done effectively. Not only are they the basis for many of the review processes which are in place (e.g., at NASA [NASA93]), but they have inspired the direction of much of the research in this area, which has tended to concentrate on improving the review *method*. Proposed improvements to Fagan's method often center on the importance and cost of the meeting. For example, researchers have proposed:

- Introducing additional meetings, such as the root cause analysis meeting of [Gilb93]
- Eliminating meetings in favor of straightforward data collection [Votta93].

Recently, however, there has been some research done to try to understand better the benefits of inspection meetings. Surprisingly, such studies have reported that, while they may have other benefits, inspection meetings do not contribute significantly to the number of defects found [Votta93, Porter95]. That is, team meetings do not appear to provide a significantly more complete list of defects than if the actual meeting had been dispensed with and the union of the individual reviewers' defect lists taken. This line of research calls into question the earlier assumption that improving the detection *method* is the best focus for research effort.

[1] The research described in this chapter was sponsored in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland, and appears in published form in [Basili96].

Thus our research is most closely related to work such as [Parnas85, Knight93, Porter95] which has argued that improved review effectiveness could best be obtained by improving the review *techniques* used by the individual reviewers. Our research is also related in another way: Once we have decided to provide procedural guidance to reviewers, then it becomes possible to direct the focus of different reviewers to different aspects of the document. We hypothesize that the reviewer will be more thorough because he or she has only to concentrate on one aspect rather than the entire document. Additionally, it is hypothesized that the techniques can be designed so that together they give coverage of the entire document. Although this idea is a common thread that runs throughout this work, different researchers have addressed the problem in different ways:

- Active Design Reviews [Parnas85] advocates multiple reviews, with each review covering only part of the document. The reviewer's technique consists of answering questions posed by the author of the document.
- Phased Inspections [Knight93] proposes multiple phases for the inspection, with each phase consisting of multiple reviewers examining the entire document but with a specific focus. Some checklists are used, but procedural guidance is provided to the reviewer by means of a series of questions (which require more than a simple yes/no answer).
- Defect-Based Reading [Porter95] is of the most immediate relevance to our work and so has already been discussed in some detail.

2.1.3 Modeling

To define the proper abstractions of the document, we first looked at the major tasks the requirements would have to support throughout the software lifecycle. The idea was to come up with a set of abstractions that would demonstrate that the requirements could fulfill each of these major tasks. This could be thought of as "modeling for use" at a very high level of abstraction. We used a simple model of the software lifecycle in our environment (Figure 4) to determine that, although there were many uses for the requirements, there were primarily three contributions that the requirements made to the development of the system. Most importantly, the requirements are used: to develop a design of the system by the designer, to test that the implemented system functions properly, and to make sure that the completed system contains the functionality which was requested by the customer. A well-designed set of requirements must be able to support all of these downstream uses.

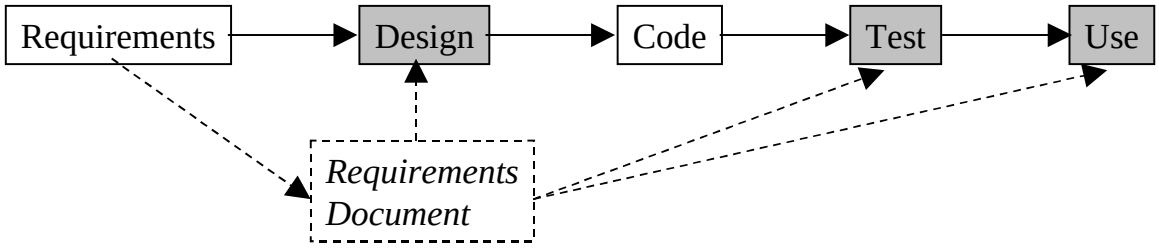

Figure 4: Simple model of requirements' use in the software lifecycle.

To ensure that a set of requirements supports all of these uses, we used each of these consumers of the document as the basis for a separate abstraction model. For example, the tester has to develop a test plan based on the information in the requirements, so one model of the system described by the requirements should be a test plan. If a test plan (at some level of detail) cannot be created from the requirements, then the requirements are insufficient since they will not be able to be used during the testing phase to develop a test plan for the system. In this way, we decided that the requirements should support three models (one for each of the important consumers we identified): a high-level design, test plan, and user's manual. Because they are handled separately in the software lifecycle, we decided to ask each reviewer to handle only one of the abstractions. We therefore anticipated review teams constructed of at least one reviewer who had been assigned each of the different abstractions.

The second crucial model, the uses of the information, was more challenging to create. To understand how developers identify defects in requirements, we first needed an idea of what a defect was. We turned to the IEEE standard on requirements [ANSI84] for a discussion of attributes that a

requirements document should possess. We then considered the lack of any of these attributes to be a defect. This led us to develop the following defect taxonomy (which is similar to that proposed in [Basili84]):

- Missing Information: (1) some significant requirement related to functionality, performance, design constraints, attributes or external interface is not included; (2) responses of the software to all realizable classes of input data in all realizable classes of situations is not defined; (3) missing sections of the requirements document; (4) missing labeling and referencing of figures, tables, and diagrams; (5) missing definition of terms and units of measures [ANSI84]
- 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: Other defects, 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.

The high-level model of finding defects in a document is therefore composed of a series of checks that the requirements possess particular attributes:

- Check completeness;
- Check clarity;
- Check consistency;
- Check correctness;
- Check conciseness;
- Check for good organization.

2.1.4 Mapping Models to Procedure

As explained in section 1.2.4, we used the interaction of the two models to produce a procedural technique (see Figure 5) by creating an abstraction procedure and a use procedure. (The "use procedure", as defined in section 1.2.4 should not be confused with the "use *perspective*", which was introduced in the last section and is specific to PBR.) That is, we developed a mechanism for checking quality attributes that could then be tailored to the techniques for creating the various abstractions.

Since this was our first time experimenting with such a technique, we did not specify a particular abstraction procedure. Instead, we let reviewers use their own procedures for creating the abstraction. We knew that our subjects were software professionals, and would have some prior experience in creating and working with each of the abstractions. Therefore, we asked them to simply use whatever procedure they were comfortable with to create a high-level abstraction of the system.

For the use procedure we were more specific. For each attribute that the requirements should have, we presented the reviewer with at least one question to answer, which addressed whether the attribute was present in the requirements or not. Obviously, the questions were tailored to the type of abstraction the reviewer was asked to develop. For example, the attribute "completeness" gives rise to a question for the tester that focuses on missing test criteria, while for the user it focuses on finding functionality which should be included in the system but is not. The use procedure thus involved answering each of the questions in turn, in order to check the document for all of the relevant attributes.

The final technique, which combines the two procedures, therefore asks reviewers to produce some physical model, which can be analyzed to answer questions aimed at uncovering defects. For example, a reviewer who is assigned the test plan abstraction would design a set of tests for a potential test plan and answer questions arising from the activities being performed. Similarly, the reviewer assigned the design plan abstraction would generate a high level design, and the reviewer assigned the user manual would sketch a manual containing the functionality involved in the system. Since each reviewer is asked to adopt the *perspective* of some downstream user of the document while looking for defects, we refer to this

family of techniques as Perspective-Based Reading (PBR). The assumption (which we test experimentally) is that the union of the perspectives provides extensive coverage of the document, yet each reviewer is responsible for a narrowly focused view of the document, which should lead to more in-depth analysis of any potential errors in the document.

These techniques are included in their entirety in Appendix B.

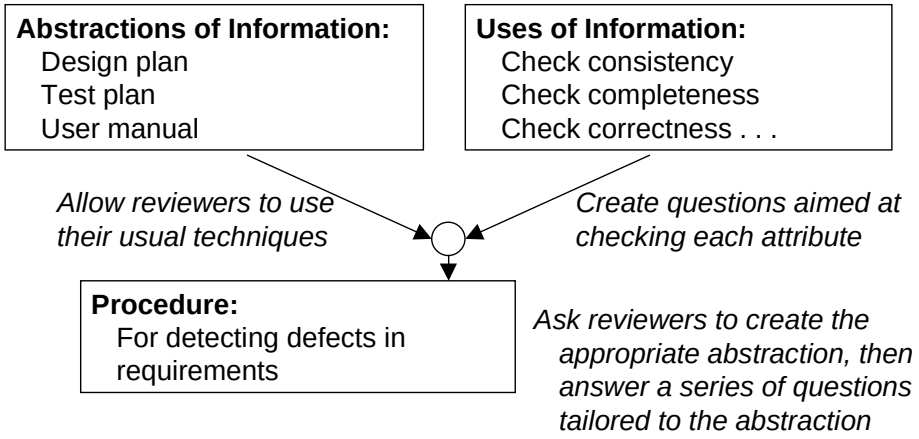

Figure 5: Building a focused technique for requirements review.

2.1.5 Description of the Study

Two runs of the experiment were conducted.[2] Due to our experiences from the initial run, some modifications were introduced in the second run. We therefore view the initial run as a pilot study to test our experimental design, and we have run the experiment once more under more appropriate testing conditions. Both runs were conducted with collaboration from the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) Software Engineering Laboratory (SEL), and so software professionals were used from this domain[3]. All subjects were volunteers so we did not have a random sample population. We accepted everyone who volunteered, and nobody participated in both runs of the experiment. Since we had to rely on volunteers, we had to provide some potential benefit to the subjects and the organization that was supporting their participation. Training in a new method provided some benefit for their time. This had an impact on our experimental design because we had to treat people equally as far as the training they received.

Our approach to evaluating the PBR family of techniques was to compare its effectiveness in uncovering defects against the method people were already using for reading and reviewing requirements specifications (which we refer to as the **usual technique**). Specifically, we tested PBR against the current SEL technique for reviewing requirements [SEL92], with which our subjects had some experience. This technique had evolved over time and was based upon recognizing certain types of concerns that were identified and accumulated as a set of issues requiring clarification by the document authors, typically the analysts and users of the system. We refer to this as a *nonsystematic technique* since although reviewers using this technique have a set of activities that they know must be undertaken and know how to approach the task of reviewing the document, there is no detailed set of steps which organizes this information and must be followed. This implies that the usual technique cannot be considered a reading technique, since it is not a set of complete, step-by-step guidelines.

[2] The author was involved only with the second run of the experiment, not with the first run (the pilot study).

[3] The SEL, started in 1976, has been developing technology aimed at improving the process of developing flight dynamics software for NASA/GSFC. This software is typically written in FORTRAN, C, C++, or Ada. Systems can range from 20K to 1M lines of source code, with development teams of up to 15 persons working over a one to two year period.

We evaluated PBR on two types of documents: requirements from the NASA flight dynamics application domain, specific to the work environment of our subjects, and "generic" requirements, which could be used to understand how generally applicable our conclusions were. For the NASA domain, two small sets of requirements derived from an existing set of requirements documentation were used. These documents, seeded with errors common to the environment, were labeled NASA_A (27 pages, 15 defects) and NASA_B (27 pages, 15 defects). For the generic application domain, two requirements documents were developed and seeded with errors. These applications were an automated parking garage control system (PG, with 16 pages and 27 defects) and an automated bank teller machine (ATM, with 17 pages and 29 defects). These applications were chosen since we reasoned our subjects would have experience using such systems, although they are outside their normal work domain. (Examples of requirements from each of the domains are included in Appendix A.) Our measure of defect detection effectiveness was the percentage of the seeded defects that was found by each reviewer.

We had originally anticipated constructing teams of three reviewers to work together during inspection meetings. Due to the constraints, however, we found this to be unfeasible. With twelve subjects, we would only have four teams that would allow for only two treatments (use of a PBR technique and the usual technique) with two data points in each. In order to achieve more statistical validity, we had each reviewer work independently, yielding six data points for each treatment. This decision not to use teams was supported by similar experiments [Porter95, Votta93], where the team meetings were reported to have little effect in terms of the defect coverage; the meeting gain was outweighed by the meeting loss. Therefore we present no conclusions about the effects of PBR team meetings in practice. We do however examine the defect coverage that can result from teams by grouping reviewers into simulated teams that combine one reviewer from each of the perspectives. We discuss this further in Section 2.1.6.

We therefore chose a design in which the technique used for review is a repeated-measures factor, in which each subject provides data under each of the treatment conditions. That is, each subject serves as its own control because each subject uses both a PBR technique and the usual technique. We hoped that this would make the experiment also more attractive in terms of getting subjects, since they would all receive similar training. At the same time, we realized that we could not have any subjects who reviewed first using PBR and then switched to their usual technique. We felt that for such subjects, their recent knowledge in PBR would have undesirable influences on the way they apply their usual technique. The opposite may also be true, that their usual technique has an influence on the way they apply PBR. However, a detailed and systematic technique, such as is used in PBR, is assumed to produce a greater carry-over effect than a non-systematic technique, such as the usual technique at NASA. An analogous decision was taken in the DBR experiment discussed in 2.1.1, where subjects starting with a systematic defect-based review technique continued to apply it in subsequent experimental tasks.

Based on the constraints of the experiment, each subject would have time to read and review no more than four documents. Since we have sets of different documents and technique to compare, it became clear that a variant of factorial design would be appropriate. (A full factorial design was inappropriate because it would require some subjects to apply systematic followed by nonsystematic techniques, and because our constraints did not allow us the time to train each subject in all three of the PBR techniques.) This leads us to the experimental design shown in Figure 6.

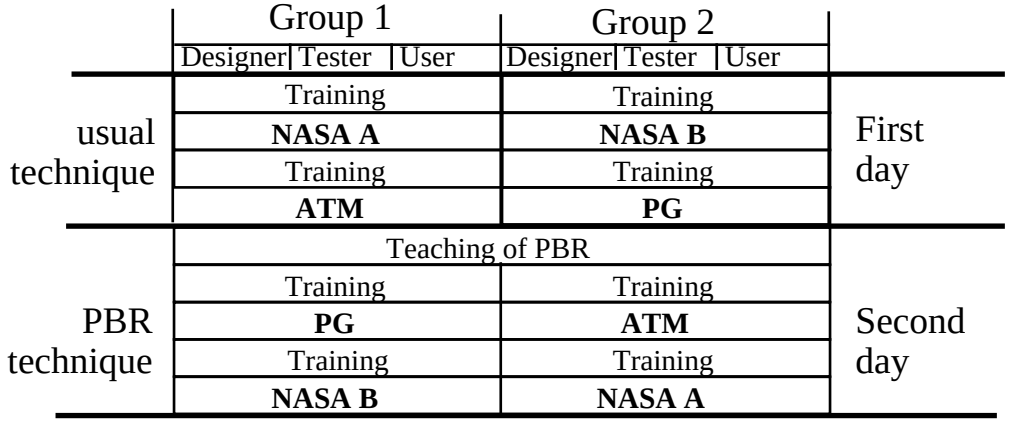

Figure 6: Design of the experiment.

The details of the statistical analysis and threats to validity can be found in [Basili96]. The important results are summarized in the following sections, organized by the pertinent evaluation question. (The list of evaluation questions was presented in section 1.3.1.) For each question, we discuss the considerations in the design that address that question, and the appropriate results from the statistical analysis. For all of the statistical tests used in this study, we used an α-value of 0.05, which means that results that are statistically significant have only a 5% chance of being due purely to chance.

2.1.6 Evaluation of Effectiveness in the Environment

We wanted to see if PBR would be effective when introduced into an organization's inspection meetings. Our question thus became, more specifically:

- If groups of individuals (such as during an inspection meeting) were given unique PBR roles, would a larger collection of defects be detected than if each reviewed the document in their usual way?

We examined the defect coverage of teams composed of one reviewer from each of the three perspectives. Because we are concerned only with the range of a team's defect coverage, and not with issues of how team members will interact, we simulated team results by taking the union of the defects detected by the reviewers on the team, and tested statistical significance by means of a Permutation Test [Edington87]. We emphasize that this grouping of individual reviewers into teams was performed after the experiment's conclusion, and does not signify that the team members actually worked together in any way. The only real constraint on the makeup of a team that applied PBR is that it contain one reviewer using each of the three perspectives; the non-PBR teams can have any three reviewers who applied their usual technique.

(Significant) (Significant)

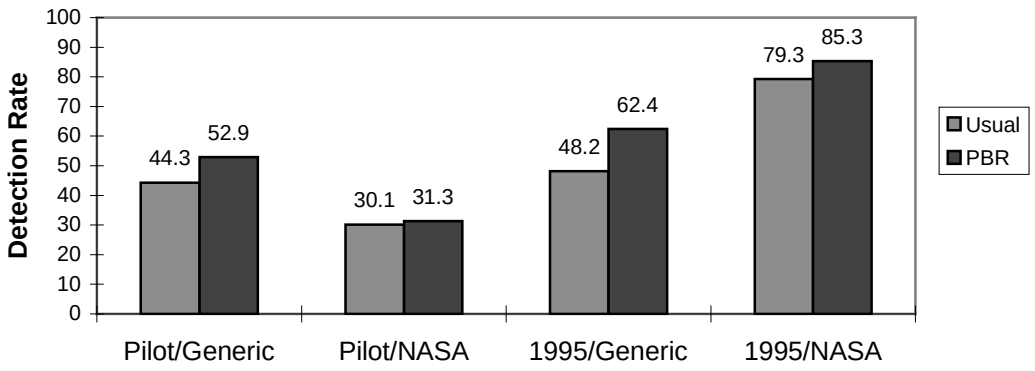

Figure 7: Simulated scores for teams using PBR and the usual technique

Figure 7 summarizes the results for team effectiveness using PBR and the usual review technique, separated by run of the experiment and the type of document (results that are statistically significant are marked). There were no significant differences seen in the pilot study. After the pilot study we made some changes to improve the experiment. In the 1995 run, the improvement due to PBR was statistically significant at the 0.05 level for both the generic and NASA documents. The reasons for this observed improvement, as compared to the pilot study, may include shorter assignments in the NASA problem domain and training sessions before each document review.

Further analysis provides an indication that, when applied to generic documents, the techniques were orthogonal, i.e. that the focus of each technique must have had some intrinsic value that prevented all of the techniques from being effective in the same way. Figure 8 presents some of the results from the 1995 experiment as typical examples of the defect coverage seen for the PBR perspectives. The numbers within each of the circle slices represent the number of defects found by each of the perspectives intersecting there. (So, for example, ATM reviewers using the design perspective found 11 defects in total: two were defects that no other perspective caught, three defects were also found by testers, one defect was also found by users, and five defects were found by at least one person from each of the three perspectives.)

Note the characteristic distribution of the defects found in each domain. In the generic domain, there were almost as many defects found uniquely by each perspective as there were found in common between perspectives, while in the NASA domain almost all defects were found by all of the perspectives.

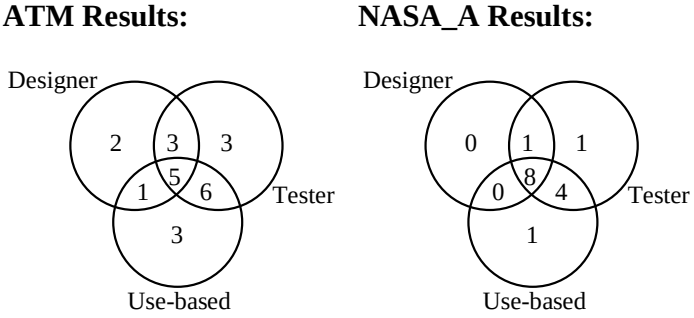

Figure 8: Defect coverage for a document from the generic (ATM) and NASA (NASA_A) domains

These indications of effective results for teams and perspectives in the generic domain have corroborated the results of the DBR study, namely, that detailed techniques can be constructed that

effectively complement one another. The practical application of this, as seen in the experiments, is that teams can be constructed which combine very specific techniques to provide improved coverage of the whole document.

2.1.7 Evaluation of Level of Specificity

It was not feasible to directly address this evaluation question in our study. We knew we would not have sufficient numbers of data points to provide our subjects with techniques at different levels of specificity and compare their results. Since the usual technique was nonsystematic, we could only compare our reading techniques against completely nonsystematic ones. This question thus becomes phrased as:

- If individuals review a document using PBR, would a larger amount of defects be found than if they reviewed the document using their usual (nonsystematic) technique?

Thus, the dependent variable for this evaluation was again the defect rate (in this case, the percentage of true defects found by a single reviewer with respect to the total number of defects in the inspected document).

This may seem like an unfair comparison – a systematic technique in which the subjects receive training from the researchers vs. a nonsystematic technique that the subjects have developed on their own. On the contrary, the subjects of the experiment have all had experience with their usual technique, and adopted it over time to be effective in their own environment. With PBR, in contrast, the subjects can all be considered novices, although we do undertake some training to minimize this.

The results for the PBR techniques and the usual NASA technique with respect to the individual performance of reviewers are shown in Figure 9. (The improvement in both domains between the pilot study and the 1995 are further confirmation of the benefit due to improving the experimental conditions.) Although in some cases, reviewers using the PBR techniques found about the same number of defects as reviewers using their usual, nonsystematic technique, PBR reviewers did perform significantly better on the generic documents in the 1995, with an improvement of about 30%. Additionally, the defect detection rate has a stronger relationship with the technique ($R^2 = 0.44$) than with any potentially confounding factors, such as the document reviewed.[4] However, no significant effects were seen in either study for the NASA domain, although Figure 9 shows that the defect detection rate for PBR reviewers was slightly higher than for reviewers using their usual technique.

These results show that under certain conditions, the use of detailed techniques can lead to improvements over nonsystematic techniques, even if the practitioner has more experience with the nonsystematic technique. Although this tells us that providing some level of detail can be better than none at all, the question of what level of detail is best in a particular environment is an important one. Several of our subjects (especially those with less experience in their assigned perspective) asked for more detailed guidance, which may indicate that the current level of detail in the technique is insufficient. We have planned another study, proposed in chapter 3, to address this question in more detail.

[4] R^2 indicates what percent of variance in the dependent variable is accounted for by the independent variable.

(Significant)

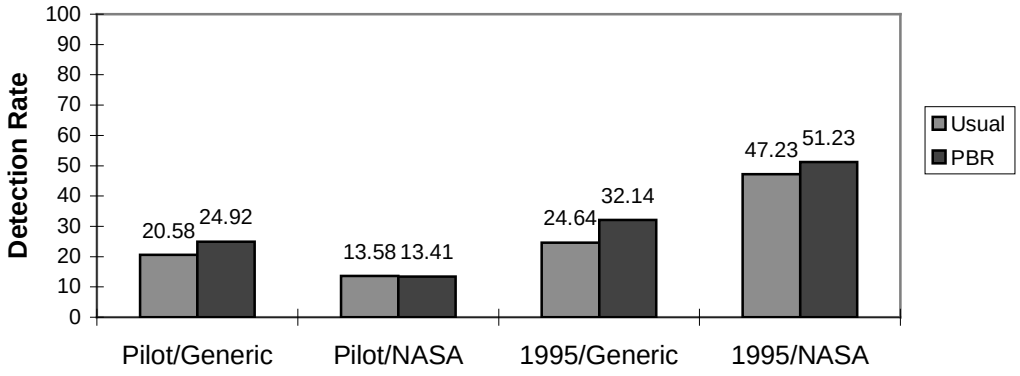

Figure 9: Mean scores for individual reviewers, for each review technique.

2.1.8 Evaluation of Measures of Subject Experience

Before the start of the experiment, we identified two ways in which we thought experience might influence the effectiveness of the techniques. First, we felt that the amount of experience a reviewer has had in reviewing requirements documents in the past would be relevant. More experience with requirements could conceivably have led the reviewer to become very accustomed to the review technique he or she had developed on their own. Relatively "novice" reviewers, on the other hand, might find it easier to learn and apply a new technique, since it might interfere less with their prior knowledge.

We considered a second relevant measure of expertise to be how much experience the reviewer had in working with the abstraction (design plan, test plan, or user manual) he or she would be asked to use in the PBR technique. Greater expertise with the abstraction could potentially effect the reviewer's comfort with the technique and the ease with which defects are spotted.

We therefore asked our subjects to fill out a background questionnaire at the beginning of the experiment. The questionnaire asked each reviewer to rate on an ordinal scale his or her level of comfort reviewing requirements documents, and to specify how many years the reviewer had spent in each of the perspective roles (designer, tester, user). We used this information to search for correlation with their individual performance using PBR.

There was no significant relationship between PBR defect rates and the reviewer's level of comfort using requirements documents. The relationship between PBR defect rates and experience with the perspective used is also weak. (Figure 10 shows the distribution for the 1995 study; the distribution in the pilot study was similar.) Reviewers with more experience do not perform better than reviewers with less experience. On the contrary, it appears that some less-experienced reviewers have learned to apply PBR better. Both Spearman's and Pearson's correlation coefficients were computed in order to measure the degree of association between the two variables for each type of document in each experiment run. In no case was there any value above 35% (values close to 100% would have indicated a high degree of correlation). A similar lack of significant relationship between past experience and the effectiveness of applying an individual process has been found by Humphrey in analyzing the experiences with teaching the Personal Software Process [Humphrey96].

Figure 10: PBR defect rate versus role experience in the 1995 run, on both generic and NASA documents

It is interesting to note that "novice" reviewers experienced some of the largest improvements in review effectiveness when using PBR. Although not a statistically significant trend, this indicates that inexperienced reviewers may be more likely to benefit from procedural guidelines. This is also born out by some of the comments reviewers made at the end of the study, on a debriefing questionnaire. Subjects with less experience seemed to follow PBR more closely ("It really helps to have a perspective because it focuses my questions. I get confused trying to wear all the hats!"), while people with more experience were more likely to fall back to their usual technique ("I reverted to what I normally do.").

2.1.9 Evaluation of Tailoring to Other Environments

We provided a lab package to enable replication of this experiment. The experimental design was well-suited to such a package since we had deliberately included two customizable considerations, which we intended to increase the desirability of a replication since it could be easily tailored to be relevant to other environments (see Figure 11).

The first was to use two separate sets of documents, to provide a baseline that could be repeated in different environments (the generics) while also providing direct feedback about PBR in a particular environment (here, the actual NASA/SEL documents). The intention is that the domain-specific documents can be varied between replications to provide experimental results that are directly applicable to a given environment. As we have seen, it also allowed us to notice differences in the way reviewers approach documents from familiar and non-familiar domains.

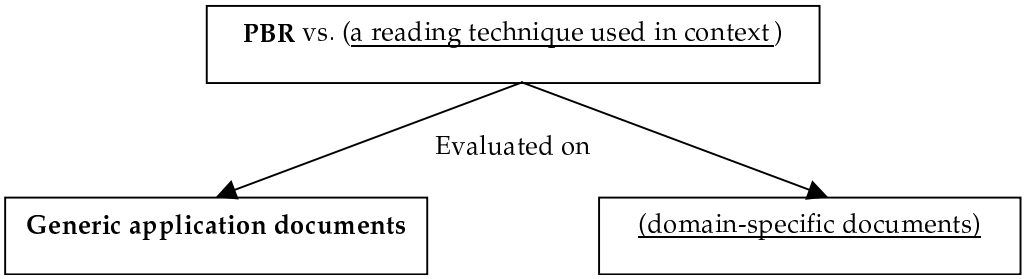

Figure 11: Customizable considerations in the design.

The second customizable design feature was the use of the usual technique. Thus this design allows us to measure the effectiveness of:

1. The usual review technique in this environment, applied to work documents from the environment (to provide a baseline for the current defect detection rate in this environment)

2. PBR, applied to work documents from the environment (to measure the change in defect detection rate if PBR were adopted in this environment)
3. The usual review technique in this environment, applied to generic documents (to understand how well this technique works on documents from outside the environment)
4. PBR, applied to generic documents (to be able to see if the effect due to PBR on different groups of subjects is comparable)

This work, and the lab package that we created for it, has been used in several replications by other researchers. (The lab package is accessible over the web at http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html.) The most directly relevant replications took place at the University of Kaiserslautern, in Germany, and the University of Trondheim, in Norway.

Two replications were run at Kaiserslautern. A novel approach of these studies was that review teams actually met in order to discuss the defects; the results concerning the team coverage of the document were not simulated, as in our studies [Ciolkowski97]. Since they achieve a similar result for their analysis of team performance, we can have greater confidence that our method of simulation was accurate. They also undertook a statistical technique called meta-analysis to combine the results of their studies and ours. Since their results were very similar to ours for teams, individuals, and perspectives, the meta-analysis allows us to have additional confidence in our results.

The replication undertaken at Trondheim is also interesting as it represents an experiment with a different but related version of the PBR techniques [Sørumgård97]. The same underlying abstractions of information and models of the low-level activities were used; however, the abstractions were mapped to procedures in a very different way. In order to be able to gauge the level of process conformance subjects used when asked to apply PBR, Sørumgård gave subjects a detailed technique for building the abstraction rather than allowing them to use their usual technique. Somewhat surprisingly, the result of this study was that the more specific technique did not result in increased process conformance.

2.1.10 Conclusions and Indications for Future Research into PBR

This study showed that the family of reading techniques that we developed could be effective at both the levels of individual reviewers and of teams. This gave us some evidence that the idea of empirically-developed techniques would be practically useful, at least in some environments. It also showed that we could define individual techniques in such a way that when applied in combination, by multiple practitioners, they could achieve specific goals (such as providing increased coverage of a particular software document).

There is still much room for improvement. Most noticeable was the failure of PBR to provide much significant benefit for documents from within the familiar work domain of our subjects. Combined with the fact that the most significant benefits from PBR were often seen for novices, we suspect that we are witnessing an interaction effect due to familiarity with the type of document being reviewed. That is, we hypothesize that when subjects are faced with documents that are similar to ones on which they have developed their own review technique, they tend to fall back on the technique that is most comfortable to them.

This is also supported by the analysis of which techniques found which defects (see Figures 8). We know that the techniques seemed to find different defects on the generic documents, but on the NASA documents reviewers from each perspective seemed to find the same defects. One explanation for this is that the reviewers weren't using the different PBR techniques at all when they reviewed the NASA documents. We also got feedback from the subjects that supported this view; several found it tempting to fall back to their usual technique when reviewing the NASA documents, thus underestimating the effect of using PBR.

We revisit this theme again in Chapter 3 when we propose future work in this area.

2.2 INVESTIGATING A TECHNIQUE FOR REUSE OF CODE AND DESIGN: SCOPE-BASED READING[5]

2.2.1 Introduction

The experiment described in this section was undertaken to understand whether the RTD process could be applied to a Construction Task in a fashion similar to that in which it was applied to an Analysis Task (discussed in section 2.1).

The techniques presented in this section were designed to address a Construction Task, the second major category of software engineering tasks (Figure 3). A Construction Task is aimed at answering the question: Given an existing artifact, how do I understand how to use it as part of a new system? Reading for construction is important for comprehending what a system does, and what capabilities exist and do not exist; it helps us abstract the important information in the system. It is useful for maintenance as well as for building new systems from reusable components and architectures [Basili96b].

We chose to focus on the understanding of *object-oriented frameworks* as the artifact to be used in system development. An object-oriented framework is a class hierarchy augmented with a built-in model that defines how the objects derived from the hierarchy interact with one another to implement some functionality. A framework is tailored to solve a particular problem by customizing its abstract and concrete classes, allowing the framework architecture to be reused by all specific solutions within a problem domain. By providing both design and infrastructure for developing applications, the framework approach promises to develop applications faster [Lewis95]. The most popular frameworks are in the GUI application domain (e.g. MacApp, ET++, CommonPoint) and in the drawing domain (e.g. HotDraw, UniDraw) but frameworks have also been developed in other domains such as multimedia, manufacturing, financial trade, and data access.

The choice to focus on frameworks was motivated primarily by two reasons:

1. Frameworks are a promising means of reuse. Although class libraries are often touted as an effective means of building new systems more cheaply or reliably, these libraries provide only functionality at a low level. Thus the developer is forced to provide the interconnections both between classes from the library and between the library classes and the system being developed. Greater benefits are expected from reusable, domain specific frameworks that usefully encapsulate these interconnections themselves.
2. Frameworks have associated learning problems that affect their usefulness. The effort required to learn enough about the framework to begin coding is very high, especially for novices [Taligent95, Pree95]. Developing an application by using a framework is closer to maintaining an existing application than to developing a new application from scratch: in framework-based development, the static and dynamic structures must first be understood and then adapted to the specific requirements of the application. As in maintenance, for a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has yet been done on minimizing this learning curve.

Since we approached this study from the viewpoint of software reading, our primary focus was on the processes developers would engage in, as they attempted to discover enough information about the framework to be able to use it effectively. We reasoned that the best approach would be to observe a number of different approaches or techniques and their effects in practice. From this information, we hoped to determine what kinds of strategies could be used, and for which situations they were likely to be particularly well- or ill-suited. Ultimately we hoped to gain an understanding of the deeper principles involved in framework usage by studying the interaction between the different techniques and the specific task undertaken.

2.2.2 Related Work

A survey of the literature on frameworks shows that relatively little has been written on *using* frameworks (as opposed to building or designing them). Most of the work on using and learning frameworks tends to

[5] The research described in this chapter was sponsored by NSF grant CCR9706151 and UMIACS. Our thanks also go to the students of CMSC 435, Fall 1996, for their cooperation and hard work.

concentrate on strategies for framework designers to use in documenting their work. [Johnson92, Beck94, Froehlich97, Gangophadyay95, Vlissides91, Frei91] are all examples of this trend (discussion of this work can be found in [Shull98].

An important weakness which is shared by all of these approaches is that they assume that the framework developer will be able to anticipate future uses of the framework adequately to provide enough patterns (or exemplars, or tutorial lessons) in sufficient detail. An alternate research approach avoids making any such assumptions about framework usage in order to undertake an empirical study of how developers go about performing the necessary tasks. Such an approach is not new, and has in fact proven useful in understanding how developers perform related tasks such as understanding code [vonMayrhauser95] or performing maintenance [Singer96]. [Codenie97] applied this approach to studying framework usage in an industrial environment and indicates that, in most cases, framework customization will be more complex than just making modifications at a limited number of predefined spots. Documentation that is based limited modifications will be too constraining and will not provide support for many realistic development problems, which far from requiring isolated changes may sometimes even require changes to the underlying framework architecture.

Our study belongs in this category of empirical study of practical framework use. It is similar in type to the study undertaken by Schneider and Repenning [Schneider95], which draws conclusions about the process of software development with frameworks from 50 application-building efforts supervised by the authors. The large number of projects followed allowed the authors to examine both successful and unsuccessful projects, and their observation of (and sometime participation in) the process allowed them to both characterize the usual process and to identify some conditions that contribute to unsuccessful projects.

2.2.3 Modeling

We began by creating two models to describe how frameworks support reuse: an abstraction model and a use model (as explained in section 1.2.4). Coming up with a suitable abstraction to represent the framework turned out to be quite difficult, as there is no consensus in the literature as to the best representation or documentation for a framework. Two main models have been presented as effective ways to learn frameworks; we decided to use each of these models as the basis for a separate technique, evaluating them by observing the strengths and weaknesses of the resulting techniques in practice.

The most common description of a framework uses the class hierarchy to describe the functionality supported by the framework and an object model to describe how the dynamic behavior is implemented. Most of the common descriptions of a framework in the literature (e.g. [Lewis95], [Taligent95]) present a model of the framework similar to this one. This abstraction forms the basis for what we refer to as the *Hierarchy-Based (HB)* technique.

As an alternative abstraction, we decided to look at the framework through a set of example applications which, taken together, were meant to illustrate the range of functionality and behavior provided by the framework. Although a detailed examination of learning frameworks by means of examples has not been undertaken, learning by example also seemed a promising approach. Sets of example applications have been used to document some frameworks (the framework we used came with such a set). The approach has also been recommended for similar types of activities, such as learning effective techniques for problem solving [Chi87], or learning how to write programs in a new programming language [Koltun83, Rosson90]. It has also been argued that learning by examples is well-suited for "domains where multiple organizational principles and irregularities in interaction exist" [Brandt97], which may be a fair assessment of the large hierarchy of classes in a framework. We refer to the technique created using this abstraction as the *Example-Based (EB)* technique.

The model of use was somewhat easier to create. Regardless of how the framework was represented, it was to be used as a source of design or code to be reused in the system being built. The use model is therefore relatively simple:

- Find functionality that is useful for the system being built.
- Adapt the code responsible for that functionality, and incorporate it into the new system.

2.2.4 Mapping Models to Procedure

We created two abstraction procedures to guide exploration through each of the abstraction models described above.

To teach subjects how to use the hierarchy-based abstraction model, we created an abstraction procedure that could be used to understand the functionality provided in the class hierarchy. As they followed this procedure, subjects would:

- Concentrate on abstract classes in the hierarchy (to provide an understanding of the broad classes of functionality);
- Iterate through deeper and deeper levels of concrete classes to find the most specific instantiation.

The example-based abstraction procedure was created in an analogous fashion, by providing steps to guide the user through different levels of detail within the abstraction model. The framework we used in this study came with a set of examples at varying levels of complexity that was constructed to demonstrate the important concepts of the framework. To help subjects learn the framework we created an abstraction procedure that would assist the subject in tracing functionality by:

- Selecting examples from the example set;
- Identifying the responsible objects in the implementation of each example;
- Identifying the responsible methods (or specific lines of code, if necessary) in each object.

We then completed the two techniques by tailoring a use procedure to the steps of each abstraction procedure (Figure 12). For example, for the example-based technique, we created versions of the use procedure for:

- Identifying which examples were likely to contain reusable functionality;
- Identifying classes within those examples that were potentially useful to the new system;
- If necessary, identifying attributes and methods within those classes that could be reused;
- Incorporating the functionality found into the new system.

Note how the first three of these use procedures each correspond with a step in the example-based abstraction procedure.

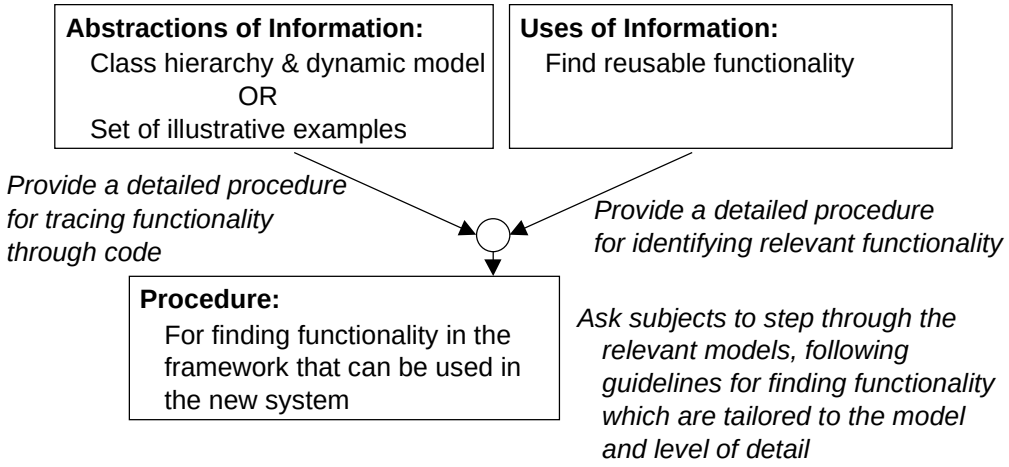

Figure 12: Producing a focused technique for reusing functionality from an OO framework

The final techniques were step-by-step guidelines that could be taught to the subjects and used to find reusable functionality. Collectively, we refer to these techniques as Scope-Based Reading (SBR), since they give the user a different scope or breadth of knowledge about the framework. Since the HB technique was focused on a search through the framework hierarchy, we assumed it might give the user a broader knowledge of the range of functionality the framework provides. The EB technique, on the other hand, was assumed to concentrate the user's focus more directly on the framework functionality that was closest to the functionality being sought at a particular time. Both techniques are included in Appendix C.

2.2.5 Description of the Study and Analysis

To undertake an exploratory analysis into framework usage, we ran a study as part of a software engineering course at the University of Maryland. Our class of upper-level undergraduates and graduate students was divided into 15 two- and three-person teams. Teams were chosen randomly and then examined to make certain that each team met certain minimum requirements (e.g. no more than one person on the team with low C++ experience). Each team was asked to develop an application during the course of the semester, going through all stages of the software lifecycle (interpreting customer requirements into object and dynamic models, then implementing the system based on these models). The application to be developed was one that would allow a user to edit OMT-notation diagrams [Rumbaugh91]. That is, the user had to be able to graphically represent the classes of a system and different types of relations between them, to be able to perform some operations (e.g. moving, resizing) directly on these representations, and to be able to enter descriptive attributes (class operations, object names, multiplicity of relations, etc.) that would be displayed according to the notational standards. The project was to be built on top of the ET++ framework [Weinand89], which assists the development of GUI-based applications. ET++ provides a hierarchy of over 200 classes that provide windowing functionality such as event handling, menu bars, dialog boxes, and the like.

Before implementation began on the project, the teams were randomly divided into two groups and each was taught only one of the two framework models and its corresponding technique for using the framework. During the implementation, we monitored the activities undertaken by the students as much as possible in order to understand if our learning techniques were used, what other learning approaches were applied, and which of these were effective. To do this, we asked the students to provide records of the activities they undertook so that we could monitor their progress and the effectiveness of the techniques, and augmented this with interviews after the project was completed.

Since the analysis was carried out both for individuals and the teams of which they were part, we were able to treat the study as an embedded case study [Yin94]. Over the course of the semester, we used a number of different methods to collect a wide variety of data (which are described in more detail in [Shull98] and summarized in Table 1). This mix of methods is in keeping with the guidelines for running observational studies in software engineering presented in [Singer96]: that, in order to obtain an accurate picture of the work involved, a variety of methods must be used at different points of the development cycle. This balances out the advantages and disadvantages of each individual method.

Aspect of Interest	**Measures**	**Form of Data**	**Unit of Analysis**	**Collection Methods**
Development Processes	Techniques used	Qualitative	team	interviews, final reports
	Tools used	Qualitative	team	interviews
	Team organization	Qualitative	team	interviews, self-assessments
	Starting point for implementation	Qualitative	team	interviews, final reports
	Difficulties encountered with technique	Qualitative	team	problem reports, self assessments, final reports
Product	Degree of implementation for each functionality	Quantitative	team	implementation score, final reports
Other Factors Influencing Effectiveness	Effort	Quantitative	team	progress reports, questionnaires
	Level of understanding of technique taught	Quantitative	individual	exam grades
	Previous experience	Quantitative	individual	questionnaires

Table 1: Types of measurements and means for collecting.

The analysis approach we used was a mix of qualitative and quantitative[6], in order to understand in detail the development strategies our subjects undertook. Our first step was to get an overview of what development processes teams had used. (By "development processes" we mean how the team had been organized, what techniques they had used to understand the framework and implement the functionality, whether they based their implementation on an example or started from scratch, and what tools they had used to support their techniques.) To this end, we performed a qualitative analysis of the explanations given by members of the teams during the interviews and final reports, and on the self-assessments. We first focused on understanding what went on within each of the teams during the implementation of the project. We identified important concepts by classifying the subjects' comments under progressively more abstract themes, then looked for themes that might be related to one another. Once we felt we had a good understanding of what teams did, we made comparisons across groups to begin to hypothesize what the relevant variables were in general. This allowed us to look for variations in team effectiveness that might be the result of differences in those key variables, as well as to rule out confounding factors. We provide an example of this type of research in [Shull98].

This process of building theories from empirical research is a recommended approach for social sciences and other fields that require the analysis of human behavior [Eisenhardt89, Glaser67, Miles79]. Although not a common method of analysis in software engineering, it has also been used to study human-centered factors in this discipline [Seaman97]. It is well suited for our purposes here because our variables of interest are heavily influenced by human behavior and because we are not attempting to prove hypotheses about framework usage, but rather to begin formulating hypotheses about this process, about which we currently know little. These hypotheses are tentative, but must be well-grounded in observation and explicitly tested in further studies.

In the following sections, we present those hypotheses that are relevant to our evaluation questions (as presented in section 1.3.1). A more complete description of the study, along with the entire set of resulting hypotheses and the threats to validity that must be considered, are found in [Shull98].

2.2.6 Evaluation of Effectiveness in the Environment

We planned to use this experiment to see if either or both techniques were effective, that is, if the techniques could be used effectively in the implementation of a reasonably sized software system. Since there has not yet been a large amount of work spent on understanding this area of framework use, our strategy was to treat this as an observational study. Subjects were taught one of the two techniques but were not required to use it. We reasoned that if the technique taught was effective, it would be used throughout the semester and we could observe its use to discover its strengths and weaknesses. If subjects abandoned the technique to use other approaches, which they knew or discovered to be more effective, we could observe this as well in order to determine what were effective strategies in this environment. We also intended to evaluate the system constructed in order to look for any correlation between the technique used by the subjects and the quality of the resulting system.

We found that teams used development processes that fell into 1 of 4 categories (Table 2). While no team used the HB technique for the entire semester (although these guidelines were partly incorporated into some of the hybrid techniques that were used), the EB technique did enjoy consistent use, both as taught and in combination with other techniques. It can be noted that even teams who were taught HB and not exposed to EB tended to reinvent a technique similar to EB on their own. (Some teams were even contrite about this. "We didn't realize at the time that this was the technique taught to the other part of the class, but it seemed the natural thing to do.") It seemed, therefore, that not only our EB technique, but example-based learning in general, was a natural way to approach learning such a complicated new system. This leads us to formulate:

[6] Qualitative data is information represented as words and pictures, not numbers [Gilgun92]. Qualitative analysis methods are those designed to analyze qualitative data. Quantitative data, on the other hand, is represented numerically or on some other discrete finite scale (i.e. yes/no or true/false) [Seaman].

HYPOTHESIS 1: Example-based techniques are well-suited to use by beginning learners.

Category	Number of Teams	Number Originally Taught EB, HB	Description
EB	5	5, 0	Students in this category used the EB technique as it was taught, following the guidelines as closely as they could.
EB/HB	5	0, 5	This is a hybrid approach that focuses on using examples to identify classes important in the implementation of a particular functionality. The main difference from the EB technique is that, in the hybrid technique, the student does not always begin tracing the functionality through the code, but may instead use the example to suggest important classes and then return to the framework hierarchy to focus on learning related classes.
ad hoc EB	4	2, 2	This was an ad hoc approach that emphasized the importance of learning the framework via examples, but ignored the detailed guidelines given. The primary difference between these techniques and the EB category is that ad hoc EB techniques are missing a consistent mechanism for selecting examples and tracing code through them.
EB/scratch	1	1, 0	The team used the EB technique to identify basic structure and useful classes, but implemented the functionality mostly from scratch.

Table 2: Description of development processes observed in the study.

In contrast, subjects tried to use the HB technique but eventually abandoned it. Qualitative analysis was necessary to understand why this happened. What was wrong with the HB technique that made it not useful in this environment? We analyzed the student remarks from the problem reports, self-assessments, and final reports to see if characteristic difficulties had been reported. A common theme (remarked upon by half of the teams who had been taught HB) was that the technique gave subjects no idea which piece of functionality provided the best starting place for implementation, or where in the massive framework hierarchy to begin looking for such functionality.

Teams also registered complaints about the time-consuming nature of the HB technique - especially compared to an example-based approach where implementation can begin much more rapidly, hierarchy-focused approaches seem to require a much larger investment of effort before any payoff is observed. One team pointed out that they had explicitly compared their progress against other (as it happened, Example-Based) teams: "We talked to other groups, and they seemed to be getting done faster with examples. So after the first week we started going to examples, too."

Despite these difficulties, students reported that they felt that the HB technique would have been very effective if they had had both sufficient documentation to support it and more time to use it. "The Hierarchy-Based technique would be helpful if you have the *time*," said one group in the final interviews, "but on a tight schedule it doesn't help at all." Another opinion was expressed by the group who said, "It's the technique I normally use anyway - and it would have been especially good here when the examples are not enough for implementing the functionality." There seemed to be a consensus that it would have allowed them to escape from the limitations of the example-based approach and engage in greater customization of the resulting system, but simply wasn't usable in the current environment. Five teams were able to create effective strategies that were hybrids of the hierarchy-based and example-based methods (EB/HB). However, the lack of guidance as to how to get started, and the time required to learn the necessary information to use it effectively, meant that no development teams used it exclusively for a significant portion of the implementation phase. (By no means was this a completely negative development, as we now have more detail on techniques that minimize that crucial learning curve.)

HYPOTHESIS 2: A hierarchy-focused technique is not well-suited to use by beginners under a tight schedule.

2.2.7 Evaluation of Level of Specificity

As discussed in the previous section, since this was intended to be an observational study, we hoped to assess whether (among the subjects who followed one of our techniques) it was more useful to follow the technique exactly as presented or to follow just the broad outline. This study presented us with just such an opportunity because some teams followed the EB technique exactly while others followed it only to a certain extent (i.e. they used an "EB-variant", the EB/HB, ad hoc EB, or EB/scratch technique). We can compare these two groups to understand if the level of specificity at which the technique is followed has an effect, and whether the technique at its current level of detail is useful.

To perform this analysis, we focused on certain *key functionalities*, that is, certain requirements for which there was a large degree of variation among teams in terms of the quality of the implementation. The quality of implementation was measured by assessing how well the submitted system met each of the original functional requirements (on a 6 point scale based upon the suggested scale for reporting run-time defects in the NASA Software Engineering Laboratory [SEL92]). The score for each functional requirement was then weighted by the subjective importance of the requirement (assigned by us) and used to compute an implementation score that reflects the usefulness and reliability of the delivered system.

We then used statistical tests to investigate whether the level of detail at which the technique was followed (EB or EB-variant) had an influence on the quality of implementation. Since both of the variables in this analysis (technique followed and result from implementation) are on a nominal scale (i.e. expressed as categories) we apply the chi-square test of probability[7]. This tests whether the proportion of teams in each type of implementation varies due to the type of technique that was used. Specifically, the null hypothesis is that the proportion of teams who achieved each type of implementation is independent of the technique that was used to perform the implementation. Due to our small sample sizes and the exploratory nature of this study, we used an α-level of 0.20, which is higher than standard levels. We also present the product moment correlation coefficient, *r*, as a measure of the effect size [Judd91]. (An *r*-value of 0 would show no correlation between the variables, whereas a value of 1 shows a perfect correlation.) We realize that these tests do not provide strong statistical evidence of any relationship, but instead see their contribution as helping detect patterns in the data that can be specifically tested in future studies.

We identified 4 key functionalities: links, dialog boxes, deletion, and multiple views. They illustrate 3 types of situations that may arise when functionality is being sought in examples:

1. **The examples don't provide all of the functionality desired.** Key functionality 1 (links) fits into this category. There was no application with functionality that explicitly addresses this requirement of the system, although one of the examples did contain similar functionality that was less sophisticated than what was needed. Almost all (4/5) of the teams who used the EB technique implemented the less sophisticated version of the functionality found in the ER application. By comparison, less than half (4/10) of the teams who used an EB-variant technique turned in the less sophisticated implementation. A chi-square test of independence was undertaken to test whether the level of sophistication achieved was dependent on the type of technique used. The test resulted in a p-value of 0.143, which is statistically significant at the selected α-level. An *r*-value of 0.38 confirms that this shows a moderate correlation [Hatcher94] between level of sophistication and type of technique. From this example we hypothesize that:

   HYPOTHESIS 3: A detailed Example-Based technique can cause developers to not go beyond the functionality that is to be found in the example set.

2. **The functionality was completely contained in (perhaps multiple) examples.** Key functionalities 2 and 3 (dialog boxes and deletion) provide evidence that the EB technique performed about the same as

[7] We base our use of the chi-square test, rather than the adjusted chi-square test, on [Conover80], which argues that the adjusted test "tends to be overly conservative."

the variant techniques in this case. There were existing applications that showed how to implement both of these requirements for the project. The difficulty was that this functionality was spread piecemeal over multiple applications and students had a hard time finding and integrating all of the functionality they needed.

About half of the class (7/15) managed to get the dialog box functionality working correctly and interfaced with the rest of the system. The use of the EB and EB-variant techniques was distributed roughly equally between the teams who did and did not get this functionality working correctly. The chi-square test here yielded a p-value of 0.714, for which the related *r*-value is 0.10. This confirms that response levels are effectively equal between the two categories.

Teams basically implemented deletion in one of three ways, of increasing sophistication. Again, the EB and EB-variant techniques seemed equally distributed among these three categories. The chi-square test for this functionality yielded a value of 1, with an associated *r*-value of 0, indicating that response rates are exactly equal regardless of the type of technique used.

From these two examples we hypothesize:

HYPOTHESIS 4: When the functionality sought is contained in the example set, Example-Based techniques will perform about the same, regardless of the level of detail provided.

3. **The examples provide a more sophisticated implementation than is required.** Key functionality 4 (views) fits here. Applications existed which satisfied the project's requirements about views. However, other applications gave an even more sophisticated implementation that allowed views to be dynamically added and deleted. All but 3 teams chose the more sophisticated implementation. 2 of these 3 teams turning in less functionality used the EB technique. The chi-square test resulted in a p-value of 0.171, which is statistically significant at the selected α-level. An *r*-value of 0.35 shows a moderate correlation between the variables and leads us to hypothesize:

HYPOTHESIS 5: When the example set contains functionality beyond what is required for the system, a sufficiently detailed Example-Based technique can help focus developers on just what is necessary.

We conclude from this analysis that the level of detail at which a technique is followed does indeed make a difference.

Teams who followed our technique exactly tended not to go far beyond what was provided by the example applications themselves. This leads us to note that in a situation in which the set of applications is sparse and does not contain the necessary functionality, following the technique with such a level of detail may not be appropriate. On the other hand, we note that these same teams spent less time "gold plating" the system, or adding functionality that is more sophisticated than necessary. From this, we can also conclude that if the set of applications is particularly large, then the level of detail called for in the EB technique would be helpful.

2.2.8 Evaluation of Measures of Subject Experience

As in the PBR study, we collected multiple measures of subject experience in related areas, by using questionnaires before the experiment. We used these measures to look for any correlation with the technique used and/or the quality of the resulting system. The areas of subject experience we measured were:

- Previous experience with the particular framework used in the study;
- Previous experience with reuse libraries (in structured or OO environments);
- Total programming experience (industrial or academic);
- Total OO programming experience (industrial or academic);
- Total C++ experience, since the framework was implemented in C++ (industrial or academic);
- Total experience with system design and analysis (structured or OO programming).

We removed one team from the analysis as an outlier, due to severe organizational difficulties that outweigh any other factor in predicting that team's performance (this data point is declared an outlier using the definition of [Ott93]). We used the Pearson correlation coefficient to measure the strength of the linear relationship between each measure of experience and implementation score (with scores close to 1 or -1 representing an exact linear relationship and scores tending to zero representing no linear relationship). The only measure of experience that was determined to have a correlation with effectiveness at implementation was the subject's total experience programming in industry (Pearson's correlation coefficient of 0.55, Figure 13). However, an R^2 value of 0.31 for the model means that industrial experience accounted for only 31% of the observed variation in the implementation score.

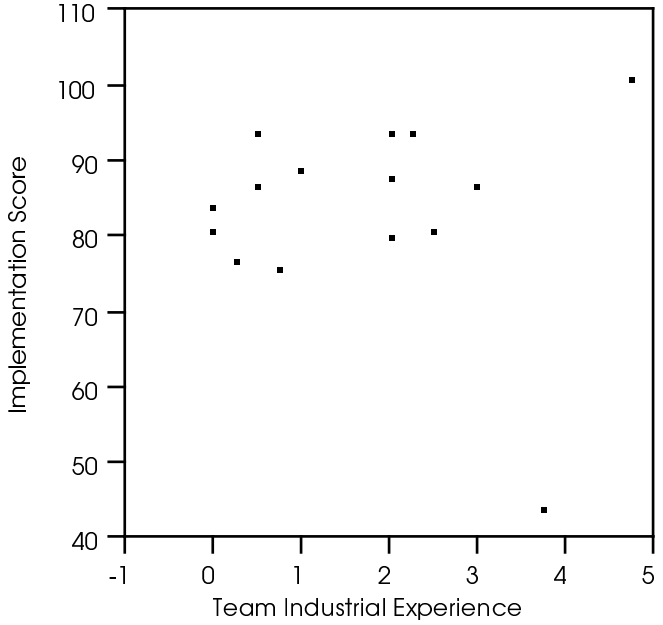

Figure 13: The total industrial experience of a team (in years) and its correlation with its effectiveness at implementation of the project (the team with implementation score of 44 has been removed from analysis as an outlier).

As we discuss in section 2.2.6, the most relevant measure of experience seemed to be familiarity with the particular framework being used, and this had a drastic effect on the course of the study. Although certain features of the HB technique might have been more effective, had they been able to be used, the technique simply wasn't well suited to beginning learners on a tight schedule.

2.2.9 Evaluation of Tailoring to Other Environments

This study has not provided a good answer to this evaluation question, largely because we are not sure about the applicability of our usual strategy of building a lab package. This is true because it is difficult to package two aspects of this study:

- **The move from controlled experiment to case study.** We had originally envisioned this study as more of a controlled experiment to assess the effectiveness of the two different models (and, by implication, of the two associated techniques). However, over the course of the study we realized that we would not be able to do a meaningful comparison of the two techniques because HB was simply not well suited for our environment. Because we were collecting a wide array of data on the process being followed, we could make the transition to a qualitative study. It is, however, difficult to know how to present the "design" of this experiment.
- **The reliance on qualitative analysis.** This was a study in which qualitative process information turned out to be much more important and interesting than quantitative data on the results. However, the qualitative analysis which played such a large role in this study is also difficult to know how to package. In a sense, it is much less structured than quantitative analysis.

2.2.10 Conclusions and Indications for Future Research into SBR

The primary result of this study was the indication that the EB technique was an effective way for developers to learn and use a framework with which they were inexperienced. However, as with the PBR family of techniques in section 2.1.11, we should not be satisfied with this result but attempt to identify any problems with the techniques used in this study. We can use this knowledge in order to suggest potential improvements that can then be empirically tested in turn, as recommended by the RTD process.

The most important complaint against the EB technique is that, since implementation relies mainly on the example set, the quality and breadth of functionality of the final system are too dependent on what is provided in the examples. Comments made by our subjects indicated that they may have recognized this limitation, but did simply not have the experience to use a technique other than EB. From these indications, we hypothesize that one way to improve the EB technique may be to integrate it with the HB technique. The best use of the EB technique may be to introduce novice users to the framework and assist them in using functionality similar to that in the examples in their new system. It is possible that, for more sophisticated functionality, EB should allow users to build up the experience necessary and then refer users to a technique such as HB.

Whether or not we can redesign the techniques to work together in this way is an important question. We have evidence from the study that applying a stage at the wrong time can be a critical error. As our study has shown, applying the wrong technique to a task can result in an incorrect or incomplete implementation of the required functionality, or a needlessly time-consuming search through the framework class hierarchy.

In the next iteration of the RTD process, in which the next versions of the techniques are tested, the empirical evaluation should first test whether the new versions of the example-based and hierarchy-based techniques are effective. A secondary goal should be to put bounds on the type of task to which each is best suited. To meet these goals, a smaller case study may be more appropriate than another controlled experiment. A study using protocol analysis (e.g. [vonMayrhauser95]) would be able to focus in more detail on the specific process developers go through when attempting to determine whether a functionality is supported by the framework and the example set. Protocol analysis would also result in a deeper understanding of what kinds of conditions lead to a situation in which developers run into difficulty with the technique. With this information it would be possible to further strengthen the techniques and to propose more detailed guidelines as to when example-based techniques should be used, and when hierarchy-focused.

2.3 SUMMARY OF COMPLETED WORK

We have now completed two empirical investigations into very different types of reading techniques. Each investigation explored reading techniques for a different type of task (both analysis and construction) and different documents. Taken together, these studies provide some initial and tentative answers to our questions for evaluating the RTD process (section 1.3.2) and highlight open questions that require further research. We organize these indications from previous studies by the process evaluation questions to which they refer.

PEQ1.1 Is RTD a feasible way to create effective reading techniques?

Both the studies of PBR and SBR have shown that effective reading techniques can be created, at least for certain contexts. The study of Scope-Based Reading (SBR) demonstrated a technique that was deemed appropriate for its context because it met the evaluation criterion of being usable for successfully implementing a semester-long programming project. In the study of detecting defects in requirements using Perspective-Based Reading (PBR), the family of reading techniques was found to be superior (according to the evaluation criteria of percentage of faults found) to the usual defect detection technique that was used in the environment.

PEQ1.2 Is RTD a feasible way to improve reading techniques?

Our studies have not yet directly addressed this question. A study is proposed in chapter 3 that applies RTD to the improvement of the PBR family of techniques. Having analyzed the PBR reading techniques in practice, we would like to use our experiences in order to improve the technique. The proposed study will investigate **whether or not a reading technique can be improved through changes to its underlying models and procedures**. That is, can we effectively manipulate the underlying models (which were created in the previous PBR studies) to produce particular desired changes in the resulting technique?

This is an important question, because the RTD process is iterative and depends on the possibility of continuous improvement from one cycle to the next. It is hoped that we can change only the relevant parts of the model for each iteration and carry these changes through so that we only have to change certain parts of the technique, rather than building a new technique from scratch each time we want to improve it. The research proposed in the next chapter addresses this question by identifying key points that can be improved about the PBR techniques (based on the experiences reported by subjects in previous studies) and tracing the effect of these improvements from the underlying models through the resulting technique.

The previous studies into both PBR and SBR have provided some indications that changes to the models do affect the resulting techniques. For example, we discussed briefly in section 2.1.1 how DBR and PBR were created for the same task (fault detection in requirements) but were tailored to different notations (a formal requirements notation versus natural language). The study of SBR addressed this point more directly by discussing two alternate abstraction models for the framework (represented by a hierarchy of classes or a set of example applications) and showing how they led to distinct reading techniques. The proposed study represents a somewhat different approach since it specifies the changes to the technique in advance and attempts to achieve them through particular changes to the models.

PEQ2 Does the RTD process build up sound knowledge about important factors affecting reading techniques?

The completed work has helped identify some of these important factors. For example, the level of specificity at which the techniques are followed was shown in the SBR study to have a definite influence on the results of applying the technique. In section 2.2.7, we discussed how characteristic differences were seen between subjects who applied the technique at different levels of specificity, and how these differences made the different levels of specificity better suited to different environments.

Our studies have also provided some indications (though no definitive quantitative evidence) that subject experience may be another important factor. In the study of SBR, inexperience with the framework was identified as a reason why the HB technique was not effective, despite subjects attempting to use it and feeling that it addressed certain limitations of the EB technique. In the PBR study, although no straightforward correlation between experience and fault detection effectiveness was found, it was noted that inexperienced reviewers had some of the highest improvement rates when they switched to PBR. But

we do not yet have significant quantitative measures of the effect of experience, perhaps because we simply have not found the right measures. There are a number of different types of experience to measure in any given environment (e.g. experience with using a particular document type, a particular notation, and the particular set of activities called for in the technique). It may also be that the most highly correlated measure is one that we have not yet thought of, or is a combination of some of the others. Additionally, several studies have demonstrated productivity differences of at least 10-to-1 among even experienced programmers [McConnell98]. If we assume that this result holds for the use of reading techniques as well, then this wide variation may be responsible for hiding any effects due to experience. Effects due to subject experience do deserve further attention, since they seem to be a major factor influencing the extent to which subjects conform to a particular reading technique. Subject comments from the PBR study indicated that inexperienced subjects may have been likely to follow the technique more closely than subjects who were experienced in reviewing requirements, who tended to prefer the way they were used to doing things.

These observations indicate that we should focus on confirming the validity of our studies. Since level of specificity is important, and may vary depending on subject experience, it is important to assess the construct validity, that is, the degree to which our expectations about the technique and level of specificity being evaluated match reality. We therefore designed the study proposed in chapter 3 to answer the question, **whether or not we can enforce a particular level of specificity in a technique?** That is, can we design techniques at a given level of detail (which must be tailored to the particular environment) and then have some confidence that subjects are actually using the technique at the proper level of specificity? This question is especially important, given the role of subject experience, since a method for enforcing a level of specificity could also be used to keep experienced developers focused on the reading technique they are given and avoid interference from their usual technique.

In the proposed study, we also address the question of internal validity: whether our study is measuring effectiveness properly. We propose a new measure of effectiveness for PBR (errors rather than faults) and attempt to better understand the relation between the two measures. It is hoped that this new measure will address some of the shortcomings of using faults as our measure of effectiveness, which we discuss in more detail in chapter 3. We hope to continue to build our confidence in the external validity by encouraging replications in other environments. This has been an effective strategy for the PBR experiments, for which replications in other environments have achieved similar results.

PEQ3 Can lessons learned from the RTD process be packaged for use in other environments?
The PBR study has already given us an excellent indication that this is true, since the lab package we created for this experiment has been used to support other replications of this experiment. More importantly, other researchers have been able to use the lab package to tailor and extend the study, which has added greatly to our knowledge about PBR (as discussed in section 2.1.9). We include the creation of a similar lab package as part of our proposed experiment in the next chapter.

3 Proposed Work

3.1 Introduction

In this chapter we propose a new empirical study in the area of software reading that addresses two of our open research questions. (See section 2.3 for a full discussion of how these questions were suggested by our efforts to provide further validation of the RTD process.) The first open question is whether or not we can effectively manipulate the models of abstractions and information uses to improve the resulting reading technique. Our approach to answering this question is to use an existing technique that we know is in need of improvement, and investigate whether we can address the areas needing improvement by making specific changes to the underlying models. Therefore, we focused this study on the PBR family of techniques, since although the techniques were effective we have identified well-defined areas that need to be improved.

As discussed in section 2.1.10, the most important aspect to improve was the interference due to prior user experience, that is, the tendency of experienced users to revert to their usual techniques when faced with familiar documents. An important feature for a new version of the techniques would be a mechanism to keep users focused on the use of the technique itself, or at least to be able to gauge the amount of conformance (as was done for the study of frameworks). If we could create such a mechanism, we could provide a proof-of-concept answer to our second open question, for process evaluation question PEQ2, as to whether we can ensure that the technique under study is being executed at a given level of specificity. Knowing the level of specificity at which readers follow the technique is necessary to make sure that our evaluation of the technique is valid. That is, we need to be sure that we are not drawing conclusions about the technique from subjects who were not executing the technique as we intended.

In order to address this question, the models of the abstractions had to be made more specific, so that there would be some kind of baseline against which conformance could be measured. The final goal of course was to use these more specific models to create a more specific procedural technique, which we reasoned would represent a likely improvement since a greater level of specificity would:

- Allow researchers to gauge how closely reviewers were actually following the technique;
- Keep experienced reviewers focused on the reading technique, rather than falling back on their usual technique;
- Provide additional guidance to inexperienced reviewers.

In section 2.1.9 we discussed a replication of the original experiment at the University of Trondheim that attempted to use a more specific technique to accomplish these same ends [Sørumgård97]. Although that experiment did not observe the effects it had hoped to see (viz. increased process conformance), we believe we have improved on their method of creating reading techniques. The differences between our technique and theirs are discussed in section 3.3 below.

As another potential improvement to the PBR technique, we also made a change to the model of uses, in order to experiment with a new criterion for the effectiveness of the technique. (Recall that criteria for effectiveness need to be set in step 2 of the RTD process.) We had found through our own experience that quantifying, classifying, and defining individual faults[8] as a measure of defect detection effectiveness

[8] Based on the IEEE standard terminology [IEEE87], we first define our terms and the relationship among these terms:

- 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.

was very difficult and possibly subjective. Often, it was possible to describe the same fault in many different ways, or associate the fault with completely different parts of the requirements, hindering discussion. It was also not uncommon to find many similar defects in the requirements and be unsure whether these represented many individual faults or were merely expressions of the same fault. 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.
- Easier to quantify defects: In our experience it has been easier to group defects according to broad categories than to decide whether individual defects are unique or specific instances of the same fault.
- More relevant measures: Counting classes of faults rather than individual faults gives a better picture of how well the system in general is specified.
- Finding of additional faults: By treating faults as isolated entities there is little possibility for using discovered faults to help find new ones; perhaps this problem could be solved by finding relations between faults that could point to further faults of the same type.

It is interesting that the number of faults detected is a widely used measure of defect detection effectiveness (e.g. [Fagan86, Votta93, Porter97, Johnson97]). In contrast, little effort has been spent on reviewing requirements documents to detect errors. In this experiment, we change the model of uses to incorporate errors as well, in order to test whether this is a useful and feasible measure for detection effectiveness.

Thus there are three main contributions to be expected from this study:

1. It provides an illustration of the iterative nature of the RTD process for developing reading techniques, which was proposed in section 1.2.4. If we can demonstrate that this cycle does provide an environment in which continuous improvement is possible, we provide more evidence as to the usefulness of the QIP-based approach and address process evaluation question PEQ1.2.
2. It represents a chance to make changes to the PBR family of techniques in response to the lessons learned from our earlier empirical studies, and to test whether these changes yield an increased practical effectiveness for the problem of defect detection in requirements.
3. It addresses the open research questions identified in section 2.3. Since both of these questions have to do with the fundamental models of the reading technique, we discuss their impacts on the PBR models in section 3.2. We then trace the resulting changes through the QIP cycle: section 3.3 discusses the new PBR techniques that result, and section 3.4 discusses the design of the new experiment to evaluate them.

3.2 Modeling

Accommodating the changes described above entailed specific changes to the models. The largest changes were done to the model of abstractions of information. We kept the idea of having three separate abstractions based on the needs of downstream users of the requirements (designer, tester, and user), since previous experiments have indicated that the perspectives together provided significant coverage of the document and each caught different faults. However, we made specific changes to increase the level of specificity of the abstractions.

In the original experiment with PBR, we asked reviewers to rely on their experience with other processes in the software lifecycle (creating high-level designs, test plans, and user manuals) and to use this experience to come up with a form of the relevant abstraction with which they were comfortable. In order to increase the specificity in this version of the techniques, we chose a specific form to use for each abstraction: data flow diagrams (as an abstraction of structured design), equivalence-partitioning test cases, and use cases.

An immediate benefit of increasing the level of specificity was that we could specify the models in such a way as to decrease the overlap between the perspectives. For example, we had originally intended

to use object-oriented design rather than structured design as the basis for the design abstraction. But, there seemed to be too many similarities between the procedure for creating the OO design and the use cases; both were centered on identifying user scenarios. So, we used a structured design technique instead, which concentrated on data flow, an attribute of the system that was not present in the other two abstractions.

Since no problems were identified with the model of information uses in the previous PBR studies, the model was mostly kept intact. This was possible because, in the absence of any literature on how to discover errors in requirements directly, we decided to use the faults discovered by a reviewer as pointers to the underlying errors that caused them. Thus the model of uses was simply augmented in order to include the extra step:

- Check completeness;
- Check clarity;
- Check consistency;
- Check correctness;
- Check conciseness;
- Check for good organization;
- Find the underlying errors that led to the faults discovered in the previous steps.

3.3 Mapping Models to Procedure

The new, more specific models for the abstractions mean that we can now supply a more specific procedure for their creation, rather than forcing the reviewers to rely on prior experience. The basis for these step-by-step procedures was easily found in the literature, since software developers often need to be trained in the creation of these models for other work practices. Additionally, the procedures can contain specific instructions for recording the abstractions built, so that researchers can use the intermediate models in order to judge process conformance.

The idea of representing the procedure for fault checking as a series of questions was also kept, although a new procedure for detecting errors had to be created. We reasoned that since faults were the manifestations of errors, faults could be used as pointers back to the error that caused them. Moreover, all of the faults caused by a particular error should be related in some way semantically, since the same basic misunderstanding was responsible for all of them. This procedure therefore asked reviewers to reason about the types of faults they had already found, finding ways in which they were related, and deciding whether there were characteristic misunderstandings about the nature of the system that could have resulted in a class of faults. This outline is not unlike the method we used for qualitative analysis (described in section 2.2.5), in that both methods begin from the observation of a number of individual facts and then try to group similar facts together, in an attempt to identify classes of phenomena and thus likely causes. We called the procedure for detecting errors "error abstraction," recognizing that errors were abstracted from the actual faults detected, and based it on the qualitative analysis method. Once categories were identified, we allowed the reviewer return to the document and use the categories as guides for detecting further faults.

A more specific outline of this part of the use procedure would therefore be:

- Understand why each of the faults previously identified represents a defect in the document.
- Identify errors by classifying the faults under progressively more abstract themes.
- For each error, return to the document and make sure you have located all of the faults for which it is responsible.

The two procedures for abstracting and using the information were then combined into the final techniques for the new version of PBR (Figure 14). This was done as before, being careful to tailor the steps of the procedure for using the information to the steps of the procedure for creating the abstractions. This was easier for the original experiment (Figure 15) as there was really only one step to the abstraction procedure in that case ("Use your usual procedure for creating the abstraction"). Because we did not know the details of the individual procedures used, the best we could provide was one version of the procedure for checking for faults, to be done throughout the abstraction step. By providing a detailed step-by-step procedure in this version, we can provide versions of the procedure for checking for faults that are well-tailored to each step of the modeling procedure (Figure 16). For example, the abstraction procedure for the

test perspective asks the reviewer to examine, in turn, inputs, equivalence sets, and test cases and their expected behavior. A version of the use procedure was created for each of these steps: for checking consistency, completeness, etc. of the inputs, of the equivalence sets, and of the test cases and descriptions of the corresponding system behavior.

We believe that it was a lack of this kind of tailoring that led the Trondheim replication of the original PBR experiment to notice no correspondence between the more specific technique and increased conformance. Although that experiment, like the one we propose, replaced the single-step abstraction procedure with a detailed multi-step procedure, the same fault detection procedure from the original experiment was used (Figure 17). The fault detection procedure was thus not tailored to the abstraction procedure. Also, the Trondheim replication did not exploit the multi-step nature of the process to provide multiple versions of the fault detection procedure, tailored to the steps of the abstraction procedure. Instead, the fault detection procedure was intended to be followed throughout the abstraction procedure, as in the original experiment.

The full techniques are included in Appendix D.

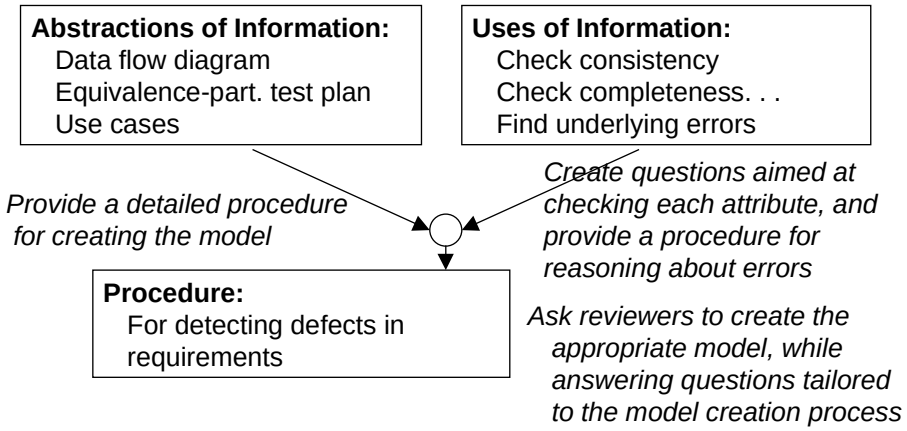

Figure 14: Building a new focused technique for requirements review. (Compare with Figure 5.)

<abstraction step 1> <steps for detecting faults, tailored to abstraction step 1>	<abstraction step 1> <steps for detecting faults, tailored to abstraction step 1> <abstraction step 2> <steps for detecting faults, tailored to abstraction step 2> . . .	<abstraction step 1> <abstraction step 2> . . . <steps for detecting faults>
Figure 15: Outline of the techniques in the original PBR study.	**Figure 16: Outline of the new version of the PBR techniques. Note that a tailored procedure for detecting faults is provided for each step of the abstraction procedure.**	**Figure 17: Outline of the PBR technique used in [Sørumgård97]. Note that the fault detection procedure is not tailored to the abstraction procedure or any of its individual steps.**

3.4 Description of the Study

We ran this experiment as a part of the Fall 1997 course CMSC735 ("A Quantitative Approach to Software Management and Engineering") at the University of Maryland, College Park. The class was composed of 66 graduate students with a wide range of experience: some students had almost no experience in industry, while others had worked as software practitioners for over twenty years. Since the class covered material such as software inspections and empirical studies in software engineering, involving the subjects in the experiment also gave them experience in both topics.

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

- Because all members of the class must be taught something, we cannot have a control group (since we cannot have a segment of the class with no treatment or training).
- Because subjects are not isolated, and would presumably discuss their experiences at various parts of the experiment amongst themselves, we cannot "cross" the experimental artifacts (that is, use a particular artifact for one group before treatment and for another group after treatment).

We therefore use an experimental design which takes the above constraints into account but which can easily be adapted for environments in which the constraints do not hold.

In this experiment, we measure the effect of teams directly rather than through simulation. This is necessary because we are unsure of how team discussion will effect the error lists. For example, we hypothesized that team discussion might affect factors such as:

- Type of errors reported: We were unsure of whether teams would report different errors than individuals, because team lists might reflect the consensus of potentially different points of view on errors. We were also unsure whether team discussion would improve the quality of errors.
- Effort: 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 that reflected each of the individual members'.

Therefore we concluded that our best strategy was to study the phenomenon observationally, as we did in the frameworks experiment in section 2.2.

We propose the following design, in which the entire group of subjects moves together through six phases, 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 (Figure 18).

	Baseline	**Individual Treatment**	**Team Treatment**	
TIME ↓	*Ad hoc* reviews			
		Error Abstraction		Document 1
			Discussion of Fault & Error Lists	
	PBR reviews			
		Error Abstraction		Document 2
			Discussion of Fault & Error Lists	

Figure 18: Design of the experiment.

For the two documents, we use the same generic documents (PG and ATM) as were used in the original PBR experiment. Since we already have a list of the *faults* they contain, we can again use the percentage of these faults found by reviewers as a measure of their detection effectiveness. There is no definitive list of the *errors* each of these documents contains, however, so we will have to analyze the errors detected by individuals in a qualitative fashion. We can also measure whether the experience of abstracting errors leads reviewers to find additional faults in the same document.

Teams were assigned so that we could test for any differences in the interaction within teams at different experience levels. Subjects were divided into three broad groups based on their amount of experience with each of the three perspectives. Teams were then composed such that all members had an

equal level of experience in the perspective they would be asked to use. 8 teams were created in which all of the members had a "high" level of experience in the perspective they were assigned, and 7 teams were created for each of the "medium" and "low" experience levels, for 22 teams altogether.

Thus for each review technique (*ad hoc* and PBR) we can collect and analyze the following process outputs:

1. The initial list of faults discovered
2. The list of abstracted errors, and a revised list of faults that may have been improved based on knowledge of the abstracted errors
3. A final list of faults and errors that result after the teams have discussed and merged the lists of the individual members

We also expect to collect detailed qualitative data about the operation of the process through questionnaires (these are included in Appendix E).

3.5 Evaluating the Reading Techniques

The above design will provide answers to our questions for evaluating reading techniques (described in section 1.3.1) through the following analyses:

EQ1. Is the technique effective in the environment?

This question can be broken down into more specific questions that are directly answered by this study:

- Does the use of PBR improve fault detection over an *ad hoc* process?
- Does the use of PBR with error abstraction improve detection effectiveness over the use of abstraction alone?
- Does the use of PBR with error abstraction require more reviewer time than the use of error abstraction alone?
- Do team meetings with PBR have a different effect than team meetings for which it has not been used?
- Is error abstraction useful for defect detection (with and without PBR)?
- Is team discussion of errors (with and without PBR) feasible? Is it useful?

Unfortunately, the design presented in section 3.4 confounds the difference due to the techniques with any difference due to the level of difficulty of the two documents. That is, if document 2 is easier than document 1, then the improvement that we would expect to see for reviewers on the second document will be impossible to separate from any improvement due to PBR. However, this situation is unavoidable given the constraints of a classroom environment, as explained in that section.

We feel that this interference will actually be small because our experience on previous studies has shown the two documents to be very close in terms of review difficulty, as measured by the detection rates observed for each document. Over the two studies run at the University of Maryland, detection rates for reviewers using their usual technique averaged 24.5% and 21.5% on the PG and ATM documents, respectively. For subjects using PBR, the averages were 27.6% and 30.8%.

We can also use a variety of measures to detect if the error abstraction portion of the technique provides any benefit. Part of this will come from a qualitative analysis of the subjects' experience, in order to detect if there were benefits added in areas we were not expecting, such as getting more familiar with the document, or being better able to repair it. We can also undertake an analysis of the number of faults discovered, to see if the experience with error abstraction uncovered new faults that had been missed by the fault detection procedure.

EQ2. Is the technique at the right level of detail?

We can obtain an answer to this question by comparing the quantitative and qualitative data for this experiment with the previous experiment in PBR, to determine if the more specific procedure does in fact provide any benefit. Since we used the same documents in both studies, we can perform a quantitative analysis of whether or not a different number of defects were found when the technique was made more specific. However, since we also collected some data in the previous study on subject satisfaction, we can also contrast this with the current subjects' satisfaction. For example, we can address important questions

such as whether the procedure, in being more detailed, was actually too constraining and less enjoyable to use. (The data collection forms and questionnaires, with which we intend to measure subjects' satisfaction with the new techniques, are found in Appendix E.)

This question is also pertinent to the error abstraction process. Because we view the abstraction process as an inherently "creative" one, in that reviewers have to make subjective decisions about what categories usefully describe the faults discovered, we do not provide a very detailed technique. We should analyze whether the use of the abstraction procedure leads to a common or similar set of errors for all reviewers, or whether the results are completely dependent on the individual reviewers (that is, every reviewer produces a significantly different list of errors). If it is the second case, then we should consider providing more detail in this process. Otherwise, one of the potential advantages of error abstraction (facilitating communication about requirements defects) would be negated.

EQ3. Do the effects of the technique vary with the experience of the subjects? If so, what are the relevant measures of experience?

As always, we will use questionnaires and background surveys to assess the experience of subjects in many related areas. Once the fault and error lists have been analyzed we will test for any correlation between these experience measures and the observed performance. (The background surveys are also included in Appendix E.)

EQ4. Can the technique be tailored to be effective in other environments?

We propose to address this question by continuing our usual approach of creating web-based lab packages, which have been described in some detail in section 1.2.4. An initial lab package has already been created and is available at:
http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/error_abstraction/manual.html.
Even though this is only a draft of the finished product, we consider this to be a promising lab package in that it has already supported replications of the experiment in a number of different environments (in Philadelphia, Italy, Brazil, and Sweden). As the analyses of these experiments are published, we will use them to build hypotheses about the larger area of reading techniques, as we have used the Kaiserslautern and Trondheim replications of the original PBR experiment.

As we continue to work on this lab package, we will use some of our lessons learned from the first lab package to improve this version. These lessons come from feedback that we received from other researchers concerning what they would have liked to see in the first PBR package. Potential changes may include:

- Specifying the experimental procedures in a way more easily used by other researchers in their own replications (by adapting the guidelines of the American Psychological Association, which has developed a standard approach for human-centered experiments in psychology);
- Incorporating the results from replications of the study directly into the lab package, so that the package can serve as a "clearinghouse" of all related knowledge on the subject;
- Including alternate file formats that are more easily modifiable by other researchers for their own replications;
- Including additional training materials that can be used to train subjects in the technique being investigated.

4 Summary

In this proposal we have addressed the practitioner's dilemma of how to choose a useful development process or tool from among the multitudes that exist. Our solution is a process for creating software reading techniques, which are procedural guidelines tailored to a particular task and environment. These techniques are developed empirically, in a way that specifies that the techniques themselves and any further improvements must be tested in practice, in order to ensure that the techniques are effective and useful. This process also gives practitioners a detailed understanding of the environment which can be used to identify sub-tasks within the technique that could benefit from tool support, so that the selection of ready-made tools or other techniques can be done with some objective, empirical basis.

We presented an outline of our process for evaluating specific software reading techniques, including particular evaluation questions to be answered for each technique created. We discussed two completed studies that we have run, as well as replications of these experiments in other environments. For each of these studies, we provided concrete examples of how reading techniques are created, by discussing the design, evaluation, and packaging of the techniques under study.

We have also used these studies to test the validity of our process for developing reading techniques, and to begin building up knowledge about reading techniques in general. Because this knowledge is at an early, provisional stage, we proposed a further study in this area. This proposed study tests the effectiveness of specific changes to one of our techniques. These changes have been suggested by previous work in this area, and have been chosen to address open questions about software reading techniques and our validation process. The analysis of the data collected during this study will demonstrate if we understand the application area well enough to make changes that provide a definite improvement to a technique in this area, and may provide evidence as to effective strategies for planning improvements to existing reading techniques. To show that this is a feasible and useful study we:

- Include the specific reading techniques being tested, and demonstrate how they were created to address questions that have arisen from previous work;
- Include the data collection forms and questionnaires, with which we collect qualitative and quantitative data as to the use of the techniques in practice;
- Show how the study addresses each of our specific evaluation questions for reading techniques.

This study will add further to our knowledge, both about this specific application area and about the process of developing reading techniques in general. The replications that are already planned in other environments by independent researchers present an excellent chance for both corroborating the evidence provided by our study as well as for understanding the influence of environmental factors on reading techniques.

References

[ANSI84] ANSI/IEEE. "IEEE Guide to Software Requirements Specifications". Standard Std 830-1984, 1984.

[Basili84] V. Basili, D. Weiss. "A Methodology for Collecting Valid Software Engineering Data". *IEEE Trans. on Software Engineering*, 10(11): 728-738, November 1984.

[Basili85] V. Basili. "Quantitative Evaluation of Software Engineering Methodology". Technical Report TR-1519, Computer Science Department, University of Maryland, College Park, July 1985.

[Basili86] V. Basili, R. Selby, D. Hutchens. "Experimentation in Software Engineering". *IEEE Trans. on Software Engineering*, 12(7): 733-743, July 1986.

[Basili88] V. Basili, H. D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environments". *IEEE Trans. on Software Engineering*, 14(6): 759-773, June 1988.

[Basili93] V. Basili. "The Experimental Paradigm in Software Engineering". In H. D. Rombach, V. Basili, R. Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions, Lecture Notes in Computer Science #706*, Springer-Verlag, August 1993.

[Basili93b] V. Basili. "The Experience Factory and its Relationship to Other Improvement Paradigms". In *Proceedings of the 4th European Software Engineering Conference*, Garmish-Partenkirchen, Germany, September 1993.

[Basili94] V. Basili, S. Green. "Software Process Evolution at the SEL". *IEEE Software*, 11(4): 58-66, July 1994.

[Basili95] V. Basili, G. Caldiera. "Improve Software Quality by Reusing Knowledge and Experience". *Sloan Management Review*, 37(1): 55-64, Fall 1995.

[Basili95b] V. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora, R. Pajerski. "Special Report: SEL's Software Process Improvement Program". *IEEE Software*, 12(6): 83-87, November 1995.

[Basili96] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, M. Zelkowitz. "The Empirical Investigation of Perspective-Based Reading". *Empirical Software Engineering: An International Journal*, 1(2): 133-164, 1996.

[Basili96b] V. Basili, G. Caldiera, F. Lanubile, F. Shull. "Studies on reading techniques". In *Proceedings of the Twenty-First Annual Software Engineering Workshop*, Greenbelt, Maryland, December 1996.

[Basili97] V. Basili. "Evolving and Packaging Reading Technologies". *The Journal of Systems and Software*, 38(1): 3-12, July 1997.

[Beck94] K. Beck, R. Johnson. "Patterns Generate Architectures". In *Proceedings of ECOOP'94*, Bologna, Italy, 1994.

[Brandt97] D. S. Brandt. "Constructivism: Teaching for Understanding of the Internet". *Communications of the ACM*, 40(10): 112-117, October 1997.

[Burgess95] A. Burgess. "Finding an Experimental Basis for Software Engineering: Interview with Victor Basili". *IEEE Software*, 12(3): 92-93, May 1995.

[Campbell63] D. Campbell, J. Stanley. *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Company, 1963.

[Chi87] M. Chi, M. Bassok, M. Lewis, P. Reimann, R. Glaser. "Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems". Technical Report UPITT/LRDC/ONR/KBC-9, University of Pittsburgh, 1987.

[Ciolkowski97] M. Ciolkowski, C. Differding, O. Laitenberger, J. Muench. "Empirical Investigation of Perspective-based Reading: A Replicated Experiment", Technical Report ISERN-97-13, April 1997.

[Codenie97] W. Codenie, K. DeHondt, P. Steyaert, A. Vercammen. "From Custom Applications to Domain-Specific Frameworks", *Communications of the ACM*, 40(10): 71-77, October 1997.

[Conover80] Conover. *Practical Nonparametric Statistics, 2nd Edition*, John Wiley & Sons, 1980.

[Deimel90] L. Deimel, J. F. Naveda. "Reading Computer Programs: Instructor's Guide and Exercises". Technical Report CMU/SEI-90-EM-3, Carnegie Mellon University, 1990.

[Edington87] E. S. Edington. *Randomization Tests.* Marcel Dekker Inc., 1987.

[Eisenhardt89] K. Eisenhardt. Building Theories from Case Study Research, *Academy of Management Review*, 14(4), 1989.

[Fagan86] M. Fagan. "Advances in Software Inspections". *IEEE Transactions on Software Engineering*, 12(7): 744-751, July 1986.

[Fenton94] N. Fenton, S. L. Pfleeger, R. Glass. "Science and Substance: A Challenge to Software Engineers". *IEEE Software*, 11(4): 86-95, July 1994.

[Fenton97] N. Fenton, S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[Frei91] C. Frei, H. Schaudt. *ET++ Tutorial: Eine Einführung in das Application Framework.* Software Schule Schweiz, Bern, 1991.

[Froehlich97] G. Froehlich, H. Hoover, L. Liu, P. Sorenson. "Hooking into Object-Oriented Application Frameworks". In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997.

[Gangopadhyay95] D. Gangopadhyay, S. Mitra. "Understanding Frameworks by Exploration of Exemplars." In *Proceedings of the 7th International Workshop on CASE*, July 1995.

[Gilb93] T. Gilb, D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[Gilgun92] J. Gilgun. "Definitions, Methodologies, and Methods in Qualitative Family Research". In J. Gilgun, K. Daly, G. Handel, editors, *Qualitative Methods in Family Research*. Sage Publications, 1992.

[Glaser67] H. G. Glaser, A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

[Hatcher94] L. Hatcher, E. J. Stepanski. *A Step-by-Step Approach to Using the SAS® System for Univariate and Multivariate Statistics.* SAS Institute Inc., 1994.

[Heninger85] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application". *IEEE Transaction on Software Engineering*, 6(1): 2-13, 1985.

[Humphrey96] W. Humphrey. "Using a Defined and Measured Personal Software Process". *IEEE Software*, 13(3): 77-88, 1996.

[IEEE87] IEEE. *Software Engineering Standards*. IEEE Computer Society Press, 1987.

[Johnson92] R. Johnson. "Documenting Frameworks with Patterns". In *Proceedings of OOPSLA '92*, Vancouver, BC, October 1992.

[Johnson97] P. Johnson, D. Tjahjono. "Assessing Software Review Meetings: A controlled experimental study using CSRS". In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, 1997.

[Judd91] C. M. Judd, E. R. Smith, L. H. Kidder. *Research Methods in Social Relations*. Holt, Rinehart and Winston, Inc., 1991.

[Kitchenham95] B. Kitchenham, L. Pickard, S. L. Pfleeger. "Case studies for method and tool evaluation". *IEEE Software*, 12(4): 52-62, July 1995.

[Knight93] J. Knight, E. A. Myers. "An Improved Inspection Technique". *Communications of the ACM*, 36(11): 51-61, November 1993.

[Koltun83] P. Koltun, L. Deimel Jr., J. Perry. "Progress report on the study of program reading". *ACM SIGCSE Bulletin*, 15(1): 168-176, February 1983.

[Law93] D. Law. "Evaluating Methods and Tools". *SQM, The British Computer Society Quality SIG*, 17, 1993.

[Lewis95] T. Lewis *et al. Object Oriented Application Frameworks.* Mannings Publication Co., 1995.

[Linger79] R. C. Linger, H. D. Mills, B. I. Witt. *Structured Programming Theory and Practice*. Addison-Wesley, 1979.

[McConnell98] S. McConnell. "Problem Programmers". *IEEE Software*, 15(2): 126-128, March 1998.

[Miles79] M. Miles. "Qualitative Data as an Attractive Nuisance: The Problem of Analysis", *Administrative Science Quarterly*, 24(4): 590-601, 1979.

[NASA93] National Aeronautics and Space Administration, Office of Safety and Mission Assurance. "Software Formal Inspections Guidebook". Report NASA-GB-A302, August 1993.

[Ott93] R. Ott. *An Introduction to Statistical Methods and Data Analysis*. Duxbury Press, 1993.

[Parnas85] D. L. Parnas, D. M. Weiss. "Active design reviews: principles and practices". In *Proceedings of the 8th International Conference on Software Engineering*, London, 1985.

[Porter95] A. Porter, L. Votta Jr., V. Basili. "Comparing detection methods for software requirements inspections: A replicated experiment". *IEEE Transactions on Software Engineering*, 21(6): 563-575, 1995

[Porter97] A. Porter, H. Siy, C. Toman, L. Votta. "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development". *IEEE Transactions on Software Engineering*, 23(6): 329-346, June 1997.

[Pree95] W. Pree. *Design Patterns for Object-Oriented Software Development.* ACM Press & Addison-Wesley Publishing Co., 1995.

[Rombach93] H. D. Rombach, V. Basili, R. Selby, editors. *Experimental Software Engineering Issues: Critical Assessment and Future Directions, Lecture Notes in Computer Science #706*, Springer-Verlag, August 1993.

[Rosson90] M. B. Rosson, J. M. Carroll, R. K. E. Bellamy. "SmallTalk Scaffolding: A Case Study of Minimalist Instruction". In *Proceedings of CHI '90*, April 1990.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Schneider95] K. Schneider, A. Repenning. "Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design Environments". In *Proceedings of the Symposium on Designing Interactive Systems*, Ann Arbor, MI, 1995.

[Seaman] C. B. Seaman, V. R. Basili. "Communication and Organization: An Empirical Study of Discussion in Inspection Meetings". Accepted for publication in *IEEE Transactions on Software Engineering*.

[Seaman97] C. B. Seaman, V. R. Basili. "An Empirical Study of Communication in Code Inspection". In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, 1997.

[SEL92] Software Engineering Laboratory. "Recommended Approach to Software Development, Revision 3". SEL report SEL-81-305, June 1992.

[Shull98] F. Shull, F. Lanubile, V. R. Basili. "Investigating Reading Techniques for Framework Learning". Technical Report CS-TR-3896, Department of Computer Science, University of Maryland, College Park, April 1998.

[Singer96] J. Singer, T. C. Lethbridge. "Methods for Studying Maintenance Activities". In *Proceedings of the 1st International Workshop on Empirical Studies of Software Maintenance*, Monterey, CA, 1996.

[Sørumgård97] S. Sørumgård. "Verification of Process Conformance in Empirical Studies of Software Development". Ph.D. thesis, Norwegian University of Science and Technology, 1997.

[Taligent95] Taligent, Inc. *The Power of Frameworks.* Addison-Wesley, 1995.

[Vlissides91] J. Vlissides. *Unidraw Tutorial I: A Simple Drawing Editor*. Stanford University, 1991.

[vonMayrhauser95] A. von Mayrhauser, A. M. Vans. "Industrial Experience with an Integrated Code Comprehension Model". *IEEE Software Engineering Journal*, pp. 171-182, September 1995.

[Votta93] L. G. Votta Jr. 1993. "Does every inspection need a meeting?". *ACM SIGSOFT Software Engineering Notes*, 18(5): 107-114, December 1993.

[Weinand89] A. Weinand, E. Gamma, R. Marty. "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework". *Structured Programming*, 10(2), 1989.

[Yin94] R. Yin. *Case Study Research: Design and Methods.* Sage Publications, 1994.

Appendix A: Sample requirements

Below is a sample requirement from the ATM document which tells what is expected when the bank computer gets a request from the ATM to verify an account:

Functional requirement 1 (From ATM document)

Description:	The bank computer checks if the bank code is valid. A bank code is valid if the cash card was issued by the bank.
Input:	Request from the ATM to verify card (Serial number and password)
Processing:	Check if the cash card was issued by the bank.
Output:	Valid or invalid bank code.

We also include a sample requirement from one of the NASA documents in order to give a picture of the difference in nature between the two domains. Below is the process step for calculating adjusted measurement times:

Calculate Adjusted Measurement Times: Process (From NASA document)

1. Compute the adjusted Sun angle time from the new packet by

$$t_{s,adj} = t_s + t_{s,bias}$$

2. Compute the adjusted MTA measurement time from the new packet by

$$t_{T,adj} = t_T + t_{T,bias}$$

3. Compute the adjusted nadir angle time from the new packet.

a. Select the most recent Earth_in crossing time that occurs before the Earth_in crossing time of the new packet. Note that the Earth_in crossing time may be from a previous packet. Check that the times are part of the same spin period by

$$t_{e-in} - t_{e-out} < E_{\max} T_{spin,user}$$

b. If the Earth_in and Earth_out crossing times are part of the same spin period, compute the adjusted nadir angle time by

$$t_{e-adj} = \frac{t_{e-in} + t_{e-out}}{2} + t_{e,bias}$$

4. Add the new packet adjusted times, measurements, and quality flags into the first buffer position, shifting the remainder of the buffer appropriately.

5. The Nth buffer position indicates the current measurements, observation times, and quality flags, to be used in the remaining Adjust Processed Data section. If the Nth buffer does not contain all of the adjusted times (and), set the corresponding time quality flags to indicate invalid data.

Appendix B: PBR procedures

B.1. Design-based Reading

Generate a design of the system from which the system can be implemented. Use your standard design approach and technique, and incorporate all necessary data objects, data structures and functions. In doing so, ask yourself the following questions throughout the design:

- Are all the necessary objects (data, data structures, and functions) defined?
- Are all the interfaces specified and consistent?
- Can all data types be defined (e.g., are the required precision and units specified)?
- Is all the necessary information available to do the design? Are all the conditions involving all objects specified (e.g., are any requirements/functional specifications missing)?
- Are there any points in which you are not clear about what you should do because the requirement/functional specification is not clear or not consistent?
- Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description/introduction?

B.2. Test--based Reading

For each requirement/functional specification, generate a test or set of tests that allow you to ensure that an implementation of the system satisfies the requirement/functional specification. Use your standard test approach and technique, and incorporate test criteria in the test suite. In doing so, ask yourself the following questions for each test:

- Do you have all the information necessary to identify the item being tested and the test criteria?
- Can you generate a reasonable test case for each item based upon the criteria?
- Can you be sure that the tests generated will yield the correct values in the correct units?
- Are there other interpretations of this requirement that the implementer might make based upon the way the requirement/functional specification is defined?
- Will this affect the tests you generate?
- Is there another requirement/functional specification for which you would generate a similar test case but would get a contradictory result?
- Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description?

B.3. Use--based Reading

Define the set of functions that a user of the system should be able to perform. Define the set of input objects necessary to perform each function, and the set of output objects that are generated by each function. This may be viewed as writing down all operational scenarios or subsets of operational scenarios that the system should perform. Start with the most obvious or nominal scenarios, and proceed to the least common scenarios or special/contingency conditions. In doing so, ask yourself the following questions for each scenario:

- Are all the functions necessary to write the operational scenario specified in the requirements or functional specifications (e.g., are all the capabilities listed in the general description specified)?
- Are the initial conditions for starting up the operational scenario clear and correct?
- Are the interfaces between functions well-defined and compatible (e.g., do the inputs of one function link to the outputs of the previous function)?
- Can you get into a state of the system that must be avoided (e.g. for reasons of safety or security)?
- Might some portion of the operational scenario give different answers depending on how a requirement/functional specification is interpreted?
- Does the requirement/functional specification make sense from what you know about the application or from what is specified in the general description?

Appendix C: SBR procedures

C.1. Example-Based Procedure

TERMINOLOGY:
Application refers to the program you are building - here, the OMT diagram editor.
Example refers specifically to one of the example applications presented with ET++, each illustrating some aspect of the framework.
Original object model is the object model of Project 1, which models the OMT editor system without taking ET++ into account.
Original dynamic model is the dynamic model of Project 2, which models the OMT editor system without taking ET++ into account.

DESCRIPTION:

1. **Based on the increments, define the functionality to be sought and find a suitable example.**

   INPUTS: Functionality of application, set of examples
   OUTPUT: Example containing functionality and set of use cases.

   a. Run the examples provided. Select for further examination the example that contains the maximum coverage of the functionality sought. If there isn't one example that contains the entire range of functionality, find the largest set of functionality that is covered by one example, and concentrate on implementing just that set for now. Record the example used and your rationale for selecting it.
   b. Compare the original object model to the example. Which classes from your original object model seem to have a corresponding object in the example? (Run the example and explore its range of functionality to determine what components are likely to be objects: What window components exist? What components respond to user events? Build use cases to illustrate the system functionality you discover.)
   c. Use your original dynamic models for the classes identified in b to determine which operations are supported by the objects in the example. Are there operations in your original dynamic models which are missing from the example?

2. **Execute and study the example selected, with respect to the functionality being sought. Build an object model of the example, selecting appropriate features.**

   INPUTS: Example and set of use cases, original object model
   OUTPUT: Event traces and object model for example

   a. Based on the increments, find the set of use cases for the functionality in this example which is relevant to the OMT system. For each use case, build an event trace that shows how the classes you identified above interact.
   b. Run the example. For each event trace:
      i. Find an object onscreen that participates in the event trace. Remember that objects in ET++ can be user-created objects, window components, etc.
      ii. "Inspect-click" on the object you have identified. This brings up a list of all objects existing at that point on the screen. Select the topmost object listed that handles the functionality, and edit its source. Now we have reached the code for a class responsible for handling a portion of the functionality sought.
      iii. Use the class definition and implementation to discover how this class handles the events you have identified in this event trace. Search through the code for this class to discover the attributes which are modified and the methods which are called; if you cannot locate a class operation which handles an event, remember to use the hierarchy browser to investigate the code for the ancestor classes as well. Does the

implementation require objects which you have not identified in your original object model? Update the event trace correspondingly if so.

iv. Using the information discovered, incorporate this class into the object model of this example. Record the attributes and operations you encounter as you trace through the code. Record relationships between classes.

v. While there are still classes in this event trace that haven't been fully documented, select another class from the event trace. Access its code either through "inspect-clicking" or directly through the code browser. Return to step iii.

vi. Once all of the relevant classes in all the event traces for all use cases are in the object model, go on to step 3.

3. Add new needed features.

INPUTS: Event traces and object model for example, original object model
OUTPUT: List of classes from example to add to application

a. Compare the object model for the example with your original object model. Determine which classes from the example will have to be added to your original model. Determine which attributes and operations are missing from your original object model that will have to be added to the class definitions.

4. Instantiate solution.

INPUTS: List of classes from example, application
OUTPUT: New version of application

a. Add new classes to your program. Modify the class definitions and implementation from the example to fit your application.
b. Add new attributes and operations to existing classes. Find the code corresponding to the initialization of the attributes and the necessary operations; modify it as appropriate to fit your application. Write the code which is necessary to extend the functionality for your application or to integrate the new functionality with the existing system.
c. Return to step 1 to implement the remaining functionality.

The following must be turned in:

1. List of examples chosen and rationales for their selection.
2. Use cases, event traces and object models for the relevant functionality of each example. The order in which you created the event traces must be indicated.

C.2. Hierarchy-Based Procedure

TERMINOLOGY:

Application refers to the program you are building - here, the OMT diagram editor.

Original object model is the object model of Assignment 1, which models the OMT editor system without taking ET++ into account.

Original dynamic model is the dynamic model of Assignment 2, which models the OMT editor system without taking ET++ into account.

DESCRIPTION:

1. Given ET++, study the class hierarchy.

INPUTS: ET++ hierarchy, ET++ architectural description
OUTPUT: List of high-level, abstract classes containing useful functionality.

a. Identify which portions of the ET++ architecture (basic building blocks, application classes, graphical building blocks, system interface layer) contain functionality that will need to be used or modified for your application.
b. Using your assessment of step a, identify abstract classes that can be used in the new application.

2. Identify classes related to the problem.

INPUTS: List of abstract classes, original object model
OUTPUT: List of ET++ classes to be used in application

a. For each class in your original object model, find the appropriate ET++ class to subclass it from. Use the hierarchy browser to help you understand how attributes and operations are inherited. Use the code viewer to examine the details of classes. Document your work as you proceed.
    i. For each abstract class you identified in 1b, examine its attributes, operations, and relations to other classes.
    ii. If the abstract class is a good match to classes in your original object model, this class can be used to subclass classes in your application. Record this class in your documentation and explain which of its attributes and operations are relevant to your application. Examine its children for an even closer match. Continue examining its siblings, or return to the parent class if there are no more siblings to examine.
    iii. If the current class contains functionality which is too specific and cannot be adapted for your application, do not consider it any longer. Record this class in your documentation and explain (either in terms of functionality or attributes/operations) why it was rejected.
b. If you cannot find an ET++ class that provides all of the operations required by a class from your object model, it is possible that your class will have to be formed as an aggregation of two or more ET++ classes. Find ET++ classes that cover subsets of the operations you need and begin thinking of ways to combine these classes.

3. Modify your original object model for the application, using classes from ET++.

INPUTS: List of ET++ classes, original object model
OUTPUT: Modified object model

a. Update your original object model to show which of your classes will be subclassed from ET++ classes (as identified in step 2a) and which will be composed of aggregates of ET++ classes (as identified in step 2b). Make sure that all relevant attributes and operations (as provided by ET++) are included.
b. If you have identified relevant classes in the ET++ class library which were not a part of the original object model, update the original model to include these classes and any appropriate links.
c. Check that the modified object model corresponds to the model of interaction provided by ET++.
d. This modified object model must be turned in.

4. Based on the modified models constructed in step 3, develop the system, exploiting the classes offered by ET++.

INPUTS: Modified object model
OUTPUT: Implementation of application

a. Develop the application, subclassing and aggregating where possible to make use of the attributes and operations offered by ET++ classes.
b. Write the code necessary to extend the functionality to the specific application, adding your own classes as needed.

The following must be turned in:
1. List of classes examined and rejected, with rationales.
2. Modified object model, showing classes provided by ET++ with their relevant attributes and operations.

Appendix D: New versions of PBR, for proposed work

Please note that the data collection forms, which are to be filled out while following these techniques, can be reached via the WWW at
http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/error_abstraction/manual.html.

D.1. Reading Technique for Structured Analysis

Use the procedure below to perform a structured analysis of the new system by creating a data flow diagram. As you work, use the questions provided to identify faults in the requirements:

1) Read through the requirements once, and then create a high-level data flow diagram. The purpose of this diagram is to clarify the system's relationship with the outside world. Identify the data sources, destinations, and stores. Then describe the processing and specify the data flows.

    a) *Data sources and destinations* are anything outside the system that originate new data or receive finished data. A source or destination could be, for example, an external system that sends or receives messages, or a department or individual that will use the system to be built. Destinations and sources are represented by closed boxes on the data flow diagram. Label as many destinations or sources as necessary on Form A.
    b) *Data stores* represent any data or documents that are stored for any length of time. This might be, for example, a pile of papers making up a file, a database on CD-ROM, or any kind of file which can be read from and written to. Data stores are represented as boxes with one side missing. Show all of the external data stores by labeling the data store symbols on Form A.
    c) Boxes with rounded corners are used to represent *processes* or activities. Again, no distinction is made between activities which are carried out by the computer and those which are performed by some other physical device, as long as the processing is part of the system. Therefore a process symbol could refer to a person undertaking a process, a computer automatically performing some data transformation, a mechanical automated process, or any other method which achieves a required result or set of results. In the high-level data flow diagram there is only one process node which summarizes the entire process. Label the process node on Form A with a description that describes the purpose of the system described by the requirements. You will use this process node as the input to step 2.
    d) Finally, the arrows in the diagram represent *data flows*. These arrows show the flow of data from a source or data store to a process in which it is used, from process to process, or from a process to a destination or data store. Label these arrows with the actual data which is provided to the system by each of the sources and stores, or which the system sends to the destinations and stores. As always, there are no distinctions due to the form of the data.

        Q1.1 From the general requirements or your knowledge of the domain, have all of the necessary data sources, destinations, and stores been defined?
        Q1.2 Can you identify what type of data is provided by or sent to each source, destination, or store? Can the data types be defined (e.g. are the required precision and units specified for the data)?
        Q1.3 Do the requirements specify the correct source, destination, or store for each of the data flows?

2) Iteratively deepen the current processing node by constructing more detailed data flow diagrams.
    a) If there is insufficient detail to construct a lower-level diagram, or if there is only one data transformation or processing step that takes place, this process does not need to have a more detailed diagram created for it.
    b) Otherwise, use Form B to construct a lower-level data flow diagram for the current processing node. First, record the name of the current node on the appropriate line in Form B. Then decompose the process into a series of smaller steps. Enter these processing nodes on Form B. You should divide the processing up amongst the new processing nodes in such a way that:
        i) Each subproblem is at the same level of detail as the other subproblems in this data flow diagram.

ii) Each subproblem can be solved as an individual problem.
iii) The solutions to the subproblems can be combined to solve the original problem.

c) Connect the processing nodes with data flow arrows to show the flow of data between the nodes. Label each arrow with the data which is being passed along.

d) The data sources, destinations, and stores from the parent node will be needed for this lower-level diagram. Record and label them on Form B (using the appropriate symbol) and draw data flow arrows to connect them to the appropriate processes. Remember to label the arrows with the appropriate form of the data.

e) Select each of the new processing nodes you created in turn and use it as the input for repeating step 2.

Q2.1 Is all of the necessary information available to create the diagram?

Q2.2 Are the requirements clear and correct about which data flows are associated with each process? (That is, is there data flowing into a process which is not needed for the process? Is a data flow missing that is necessary for the process? Is data flowing out of a process that could not have been produced by that process?)

Q2.3 Does the flow of data occur in a logical and correct manner? Can the input to each process be recognized as the output of another process or a data source or store?

Q2.4 Is a single unique process, data source, or store described as the source of each data flow?

3) When you have expanded all of the processing nodes to an appropriate level of detail, examine the completed data flow diagram and answer the following questions:

Q3.1 Does each data flow arrow have a name associated with it? Is the data it represents defined at an appropriate level of detail in the requirements?

Q3.2 Have all of the data sources, stores, and destinations listed on Form A been used in at least one of the lower-level diagrams?

Q3.3 Can you trace through the data flow diagram using actual cases and achieve the expected result?

Q3.4 Does the flow of data within each of the lower-level diagrams make sense from what you know about the application or from what is specified in the general description?

Acknowledgements
These guidelines owe much to the following works:

Foad, K. R. Cardiff ComSc Documentation Project 1996/97.
http://www.cm.cf.ac.uk/User/K.R.Foad/project/index.html.
Mynatt, Barbee Teasley. (1990) *Software Engineering with Student Project Guidance*. Prentice-Hall.
Yourdon, E., and L. Constantine. (1979) *Structured Design*. Prentice-Hall.

D.2. Reading Technique for Equivalence Partition Testing

For each requirement, generate a test or set of test cases that allow you to ensure that an implementation of the system satisfies the requirement. Follow the procedure below to generate the test cases, using the questions provided to identify faults in the requirements:

4) For each requirement, read it through once and record the number and page on the form provided, along with the inputs to the requirement.
   - **Q1.1 Does the requirement make sense from what you know about the application or from what is specified in the general description?**
   - Q1.2 Do you have all the information necessary to identify the inputs to the requirement? Based on the general requirements and your domain knowledge, are these inputs correct for this requirement?
   - Q1.3 Have any of the necessary inputs been omitted?
   - Q1.4 Are any inputs specified which are not needed for this requirement?
   - Q1.5 Is this requirement in the appropriate section of the document?

   a) For each input, divide the input domain into sets of data (called *equivalence sets*), where all of the values in each set will cause the system to behave similarly. Determine the equivalence sets for a particular input by understanding the sets of conditions that effect the behavior of the requirement. You may find it helpful to keep the following guidelines in mind when creating equivalence classes:

      - If an input condition specifies a range, at least one valid (the set of values in the range) and two invalid equivalence sets (the set of values less than the lowest extreme of the range, and the set of values greater than the largest extreme) are defined.
      - If an input condition specifies a member of a set, then at least one valid (the set itself) and one invalid equivalence sets (the complement of the valid set) are defined.
      - If an input condition requires a specific value, then one valid (the set containing the value itself) and two invalid equivalence sets (the set of values less than, and the set greater than, the value) are defined.

      Each equivalence set should be recorded in the spaces provided on the form under the appropriate input.
      - Q2.1 Do you have enough information to construct the equivalence sets for each input? Can you specify the boundaries of the sets at an appropriate level of detail?
      - Q2.2 According to the information in the requirements, are the sets constructed so that no value appears in more than one set?
      - Q2.3 Do the requirements state that a particular value should appear in more than one equivalence set (that is, do they specify more than one type of response for the same value)? Do the requirements specify that a value should appear in the wrong equivalence set?

   b) For each equivalence set write test cases, and record them beneath the associated equivalence set on the form. Select typical test cases as well as values at and near the edges of the sets. For example, if the requirement expects input values in the range 0 to 100, then the test cases selected might be: 0, 1, 56, 99, 100. Finally, for each equivalence set record the behavior which is expected to result (that is, how do you expect the system to respond to the test cases you just made up?).
      - Q3.1 Do you have enough information to create test cases for each equivalence set?
      - Q3.2 Are there other interpretations of this requirement that the implementor might make based upon the description given? Will this affect the tests you generate?
      - **Q3.3 Is there another requirement for which you would generate a similar test case but would get a contradictory result?**
      - Q3.4 Can you be sure that the tests generated will yield the correct values in the correct units? Is the resulting behavior specified appropriately?

Acknowledgements

These guidelines are based on information found in:

Jacobson, Ivar, *et al.* (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company.

Stacey, Deborah A. *Software Testing Techniques*. 27-320 Software Engineering Lecture Notes, University of Guelph. http://hebb.cis.uoguelph.ca/~deb/27320/testing/testing.html#1.4.2

D.3. Reading Technique for Use Case Construction

Create use cases to document the functionality that users of the system should be able to perform. This will require listing the functionality that the new system will provide and the external interfaces which are involved. You will have to identify actors (sets of users) who will use the system for specific types of functionality. Remember to include all of the functionality in the system, including special/contingency conditions. Follow the procedure below to generate the use cases, using the questions provided to identify faults in the requirements:

5) Read through the requirements once, finding participants involved.

   a) Identify the participants in the new requirements. Participants are the other systems and the users that interact with the system described in the requirements – that is, they participate in the system functionality by sending messages to the system and/or receiving information and instructions from it. List these participants on Form A. You may use the following questions to help you identify participants:
      - Which user groups use the system to perform a task?
      - Which user groups are needed by the system in order to perform its functions? These functions could be its major functions, or secondary functions such as system maintenance and administration.
      - Which are the external systems that use the system in order to perform a task?
      - Which components or hardware are managed by the system?
   b) Decide which of the participants are actors. Actors represent classes of users which are external to the system and interact with it in characteristic ways. Indicate these actors on the form by checking the box next to the appropriate participants. The following guidelines may be helpful:
      - Actors represent *classes* of users, and may not be the same as the *individual* users of the system.
      - Actors represent the set of external things that need to exchange information (or otherwise interact) with the system.
      - Actors are different from other participants in that their actions are non-deterministic.

   Q1.1 Are multiple terms used to describe the same participant in the requirements?
   Q1.2 Is the description of how the system interacts with a participant inconsistent with the general requirements? Are the requirements unclear or inconsistent about this interaction?
   Q1.3 Have necessary participants been omitted? That is, does the system need to interact with another system, a piece of hardware, or a type of user which is not described?
   Q1.4 Is an external system or a class of "users" described in the requirements which does not actually interact with the system?

6) Read through the requirements a second time, identifying the product functions.

   a) Identify low-level functionality in the requirements. Jot these functionalities down on a piece of scrap paper.
   b) Group functionality together to form higher-level tasks. Each of these high-level tasks will be a use case or scenario that represents, at a high-level, what kind of system functionality will be used by an actor. You may find it helpful to consider the following guidelines when group functionality together to create use cases:
      i) Group functionalities that are involved with the same subsystem.
      ii) Identify processing phases – is there a sequence or ordering to the functionality? Group together functionality which is consecutive.
      iii) Group functionalities that lead to a definite goal state for the system or for an actor.
   c) For each group of functionality, decide how the lower-level pieces are related. On a copy of Form B, use labeled boxes to represent the lower-level pieces in a particular group. Draw arrows between the boxes to identify the flow of system control ("First this functionality happens, then

this…") and use branching arrows to signify places where the flow of control could split. Remember to include exception functionality in the use case. Check the appropriate functional requirements to ensure that you have the details of the processing and flow of control correct.

d) For each use case you create, remember to signify the actor who would use the functionality, as well as the action that starts the functionality (e.g. "selecting option 2 from the main menu" might start a use case for deleting records from a database). There are spaces for recording both of these pieces of information on Form B. Finally, give the use case a descriptive name that conveys the functionality it represents. Enter the name both on the associated Form B as well as on Form A.

Q2.1 Are the start conditions for each use case specified at an appropriate level of detail?
Q2.2 Has any functionality that should appear in a use case been omitted?
Q2.3 Has the functionality been described sufficiently so that you understand how the low-level functionalities are related to each other? Does the description allow more than one interpretation as to how the functionalities are related?
Q2.4 Have necessary functionalities been omitted, according to your domain knowledge or the general description?

7) Match the actors to all of the use cases in which they participate. (Remember that if two actors participate in all of the same use cases, they probably represent a single unique actor and should be combined.) Use the third column on Form A to cross-reference the actors with the appropriate use case from the list.

Q3.1 Is it clear from the requirements which actors are involved in which use cases?
Q3.2 **Based on the general requirements and your knowledge of the domain, have all of the necessary use cases been specified for each actor?**
Q3.3 Are use cases described for an actor which are incompatible with the actor's description in the general requirements?
Q3.4 Have necessary actors been omitted (that is, are there use cases for which no actor is specified)?

Acknowledgements
These guidelines owe much to the following works:

Andersson, Michael, and Johan Bergstrand. (1995) *Formalizing Use Cases with Message Sequence Charts*. Masters Thesis for the Department of Communication Systems at Lund Institute of Technology.
Jacobson, Ivar, *et al.* (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company.

And to the work of Curt Zimmerman during his stay with the Empirical Software Engineering Group.

Appendix E: Data collection forms and questionnaires, for proposed work

E.1. Background Questionnaire

Name _____

Please, grade your experience with respect to the following 5-point scale

1 = none
2 = studied in class or from book
3 = practiced in a class project
4 = used on one project in industry
5 = used on multiple projects in industry

Analysis

- Have you experience in requirements analysis? 1 2 3 4 5
- Have you experience in use-case analysis technique? 1 2 3 4 5

Design

- Have you experience in software design? 1 2 3 4 5
- Have you experience in Structured Design? 1 2 3 4 5
- Have you experience in Object-Oriented Design? 1 2 3 4 5

Testing

- Have you experience in functional testing? 1 2 3 4 5
- Have you experience in equivalence partitioning technique? 1 2 3 4 5

E.2. Data collection form/Questionnaire for first ad hoc session

Name:
Document reviewed:

List the faults you discovered below, specifying the page number and the requirement in which each is found. An example is provided. Take as much space as you need, and separate each fault with a blank line. Remember to keep a copy of this information for yourself!

When done, skip down to the end and answer the questions there.

Fault #	Page #	Requirement #	Desc.
0	3	FR5	Since this isn't a real fault we'll assign it a number of 0. Number your faults starting from 1, of course. This fault would have been found in functional requirement 5 on page 3. If you cannot specify the requirement number (for example, some faults may affect a whole section of the document) leave this field blank. In the description you need to explain exactly what the fault is so that it could be corrected.

Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

1. How much time did you take to find faults (report just the time spent on looking for faults, not including interruptions, breaks, or other non-related activities)?

2. What percentage of the faults in the document do you think you found?

E.3. Data collection form/Questionnaire for second ad hoc session

Name:
Document reviewed:

This form has three sections. You must list the errors resulting from the abstraction process in the first section, and the revised list of faults in the document in the second. In the third section you should answer the questions which are provided. Remember to keep a copy of this information for yourself!

ERRORS
List the errors you discovered below, specifying which faults from your original fault list (from Assignment 2A) result from each of the errors. Begin numbering your errors at 1, of course. An example from the handout is provided. Take as much space as you need, and separate each error with a blank line.

Error #	Associated Faults	Error Desc.
1	3, 9	Misunderstanding of the rental logging process.

FAULTS
List only the new faults you discovered below, specifying the page number and the requirement in which each is found. Also state which error from the error list led you to the new faults. The examples from the handout are shown to illustrate how to enter the information – make sure to delete the examples before adding your faults. Continue numbering your new faults from wherever you left off on the original fault list. Take as much space as you need, and separate each fault with a blank line.

Fault #	Page #	Requirement #	Related Error	Desc.
12	11	FR14	E1	The fields and values which are to be updated are never specified.

QUESTIONS
Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

3. How much time did you take to do the abstraction process (report just the time spent on the process, not including interruptions, breaks, or other non-related activities)?

4. How much time did you spend going back to the document and identifying more faults (again, just report the time spent on looking for faults)?

5. What percentage of the faults in the document do you think you found?

6. What percentage of the errors in the document do you think you found?

7. Please assess the training that was provided for the abstraction process (mark one with an X):

__ not enough
__ enough
__ too much
Can you think of anything we could change that would have made the training sessions more effective?

8. Did the abstraction process help you identify errors in the requirements (mark one with an X)?
__ no help at all
__ some help
__ a lot of help

9. How confident do you feel that the errors you wrote down in section 1 represent real misunderstandings in the document you reviewed?
__ not confident at all; they are probably arbitrary
__ no opinion
__ very confident; they were probably real errors that whoever wrote the requirements made

10. Did you follow the abstraction process completely?
__ not at all
__ partially
__ completely

11. Did you have any difficulties or frustrations when you tried to use the abstraction process?

12. Would you recommend anything for improving the abstraction process?

13. Did the knowledge of the errors help you find additional faults in the document?
__ not at all
__ partially
__ completely

14. Do you feel the effort spent to abstract the errors was well-spent? Why or why not?

15. What did you think about the size of the document you reviewed?
__ too small
__ OK
__ too large

16. Was the application domain of the document appropriate?

17. Any additional comments?

E.4. Data collection form/Questionnaire for third ad hoc session

Team:
Document reviewed:

This form has three sections. You must list the final list of errors in the first section, and the final list of faults in the second. **Only one fault and error list must be submitted for each team!** In the third section you should answer the questions which are provided. **Each team member should submit a completed section 3.**

ERRORS
List the errors you discovered below, specifying which faults from the list below result from each of the errors. Take as much space as you need, and separate each error with a blank line.

Error #	Associated Faults	Desc.

FAULTS
List the faults you discovered below, specifying the page number and the requirement in which each is found. Also state which error from the error list led you to each fault. Take as much space as you need, and separate each fault with a blank line.

Fault #	Page #	Requirement #	Related Error	Desc.

QUESTIONS
Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

18. How much time did you take as a team to decide on the final lists (report just the time spent on discussing the lists, not including interruptions, breaks, or other non-related activities)?

19. Did you feel that the individual fault and error lists were similar or rather different? Did this fact effect the amount of time you spent on discussion?

20. When you merged the individual lists into the team list, did you find it most helpful to:
    __ merge faults first, and then discuss the associated errors
    __ merge faults first, and then re-abstract the errors
    __ merge errors first, and then discuss the associated faults
    __ other (please explain)

21. Due to meeting as a team, do you feel there were any resulting benefits to the fault and error lists (e.g. eliminated redundant faults and errors, improved descriptions, removed entries that weren't really faults)?

22. Do you feel that the error list you created as a team would be more or less useful for repairing the document than the error lists you created individually? Use the checklist to answer and then explain your reasoning below.

The individual error lists were:
__ more useful than the team version
__ not significantly different from the team version
__ less useful than the team version

23. Due to meeting as a team, do you feel there were any benefits to you as reviewers? Do you feel you understand the document better now that you have seen your teammates' work?

24. Due to meeting as a team, do you think you could do a better job of fixing the document than if you had only worked individually? Use the checklist to answer and then explain your reasoning below.

    The team meeting:
    __ would help if I had to repair the requirements
    __ wouldn't help if I had to repair the requirements

25. What percentage of the faults in the document do you think were found by the team?

26. What percentage of the errors in the document do you think were found by the team?

27. Any additional comments?

E.5. Data collection form/Questionnaire for first PBR session

Name:
Document reviewed:

List the faults you discovered below, specifying the page number and the requirement in which each is found. If you can, specify the question from the PBR technique which helped you identify the fault (just leave the column blank if you didn't really use a specific question or if you found it without using the technique). Take as much space as you need, and separate each fault with a blank line. Remember to keep a copy of this information for yourself!

When done, skip down to the end and answer the questions there.

Fault #	Page #	Requirement #	Question	Desc.

Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

28. How much time did you take to find faults (report just the time spent on looking for faults, not including interruptions, breaks, or other non-related activities)?

29. What percentage of the faults in the document do you think you found?

30. Was there enough training for the technique you were given?
    __ not enough
    __ enough
    __ too much

31. How closely did you follow the technique you were given?
    __ very closely; step by step
    __ tried to follow it informally, but didn't keep it in mind at all times
    __ ignored completely

32. How closely did you focus on the questions you were given to help find faults?
    __ very closely; tried to answer them explicitly at the appropriate point in the procedure
    __ read them several times and kept them in mind at all times
    __ ignored completely

33. How did you feel about the level of specificity in the technique you were given? Check any that apply (may be more than one):
    __ It helped me concentrate on what was important
    __ Such a high level of specificity made the process unpleasant to use

34. Did the technique you were given help you find faults?
    __ gave no help at all
    __ gave some help
    __ gave a lot of help

35. Did the technique miss anything that would have been important for that particular perspective?

36. Was the technique you were given especially familiar or unfamiliar to you? Did that make it harder or easier to use?

37. Were the forms you were given to fill out as part of the technique helpful?
    __ they actually made it harder to use the process and find faults
    __ they didn't really affect the process at all
    __ they were very helpful for following the process
    Why?

38. Any additional comments?

E.6. Data collection form/Questionnaire for second PBR session

Name:
Document reviewed:

This form has three sections. You must list the errors resulting from the abstraction process in the first section, and the revised list of faults in the document in the second. In the third section you should answer the questions which are provided. Remember to keep a copy of this information for yourself!

ERRORS
List the errors you discovered below, specifying which faults from your original fault list (from Assignment 2A) result from each of the errors. Begin numbering your errors at 1, of course. An example from the handout is provided. Take as much space as you need, and separate each error with a blank line.

Error #	Associated Faults	Error Desc.
1	3, 9	Misunderstanding of the rental logging process.

FAULTS
List only the new faults you discovered below, specifying the page number and the requirement in which each is found. Also state which error from the error list led you to the new faults. The examples from the handout are shown to illustrate how to enter the information – make sure to delete the examples before adding your faults. Continue numbering your new faults from wherever you left off on the original fault list. Take as much space as you need, and separate each fault with a blank line.

Fault #	Page #	Requirement #	Related Error	Desc.
12	11	FR14	E1	The fields and values which are to be updated are never specified.

QUESTIONS
Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

39. How much time did you take to do the abstraction process (report just the time spent on the process, not including interruptions, breaks, or other non-related activities)?

40. How much time did you spend going back to the document and identifying more faults (again, just report the time spent on looking for faults)?

41. What percentage of the faults in the document do you think you found?

42. What percentage of the errors in the document do you think you found?

43. Did the abstraction process help you identify errors in the requirements (mark one with an X)?

__ no help at all
__ some help
__ a lot of help

44. How confident do you feel that the errors you wrote down in section 1 represent real misunderstandings in the document you reviewed?
__ not confident at all; they are probably arbitrary
__ no opinion
__ very confident; they were probably real errors that whoever wrote the requirements made

45. Did you follow the abstraction process completely?
__ not at all
__ partially
__ completely

46. Did you have any difficulties or frustrations when you tried to use the abstraction process?

47. Would you recommend anything for improving the abstraction process?

48. Did the knowledge of the errors help you find additional faults in the document?
__ not at all
__ partially
__ completely

49. Do you feel the effort spent to abstract the errors was well-spent? Why or why not?

50. What did you think about the size of the document you reviewed?
__ too small
__ OK
__ too large

51. Was the application domain of the document appropriate?

52. Were there any differences from Assignment 2B in using the abstraction process? Did the fact that the fault list from Assignment 4A resulted from applying a specific procedure make it any easier or harder to discover errors?

53. Any additional comments?

E.7. Data collection form/Questionnaire for third PBR session

Team:
Document reviewed:

This form has three sections. You must list the final list of errors in the first section, and the final list of faults in the second. **Only one fault and error list must be submitted for each team!** In the third section you should answer the questions which are provided. **Each team member should submit a completed section 3.**

ERRORS
List the errors you discovered below, specifying which faults from the list below result from each of the errors. Take as much space as you need, and separate each error with a blank line.

Error #	Associated Faults	Desc.

FAULTS
List the faults you discovered below, specifying the page number and the requirement in which each is found. Also state which error from the error list led you to each fault. Take as much space as you need, and separate each fault with a blank line.

Fault #	Page #	Requirement #	Related Error	Desc.

QUESTIONS
Please answer the following questions. Your answers will be used to help us better understand the process, and will not impact the grading of this assignment:

54. How much time did you take as a team to decide on the final lists (report just the time spent on discussing the lists, not including interruptions, breaks, or other non-related activities)?

55. Did you feel that the individual fault and error lists were similar or rather different? Did this fact effect the amount of time you spent on discussion?

56. Did the fact that all members of the team used a different perspective in finding faults in Assignment 4A have an effect on how easy it was to merge the lists?
    __ the different perspectives on the team made it harder to merge
    __ there was no difference due to the perspectives
    __ the different perspectives on the team made it easier to merge

57. Did the fact that all members of the team used a different perspective have an effect on how the meeting itself was run? For example, because of the different perspectives, did all three members have more equal input to the discussion? Did team members feel less able to discuss aspects of the document that their perspective did not deal with? Please describe any related issues you encountered.

58. When you merged the individual lists into the team list, did you find it most helpful to:

__ merge faults first, and then discuss the associated errors
__ merge faults first, and then re-abstract the errors
__ merge errors first, and then discuss the associated faults
__ other (please explain)

59. Due to meeting as a team, do you feel there were any resulting benefits to the fault and error lists (e.g. eliminated redundant faults and errors, improved descriptions, removed entries that weren't really faults)? Were these benefits effected by having different perspectives for each of the team members, or were these the same as in Assignment 2C?

60. Do you feel that there were benefits to actually meeting as a team and interacting with your teammates, which could not have been achieved if you simply reviewed your teammates' fault and error lists?

61. Do you feel that the error list you created as a team would be more or less useful for repairing the document than the error lists you created individually? Use the checklist to answer and then explain your reasoning below.

    The individual error lists were:
    __ more useful than the team version
    __ not significantly different from the team version
    __ less useful than the team version

62. Due to meeting as a team, do you feel there were any benefits to you as reviewers? Do you feel you understand the document better now that you have seen your teammates' work?

63. Because of the team meeting, do you feel you understand what your teammates' perspectives involved? Do you think that they understand yours? If the answer to either of these questions is yes, has this information been beneficial or helpful in any way?

64. Due to meeting as a team, do you think you could do a better job of fixing the document than if you had only worked individually? Use the checklist to answer and then explain your reasoning below.

    The team meeting:
    __ would help if I had to repair the requirements
    __ wouldn't help if I had to repair the requirements

65. What percentage of the faults in the document do you think were found by the team?

66. What percentage of the errors in the document do you think were found by the team?

67. Any additional comments?

E.8. Final Questionnaire

1 The average amount of time spent reviewing the documents in this class was much higher than in previous experiments (and usually much higher than the 2-3 hours that we recommended).
  A Did you keep working on the assignment until you were happy with the number of faults found, or until you spent what you thought was a reasonable amount of time?

B Did you end up doing a lot of ad hoc reviewing after you had applied the other techniques, in order to find more faults? Is this mainly because the assignment was to be graded?

C Did we just really do a bad job of estimating how long it would take to apply the PBR and abstraction techniques?

2 Do you think you would have done things differently if you weren't being graded on Assignments 2 and 4?

A Would it have been better if we had announced the grading scheme ahead of time, or used a different grading scheme?

3 The questionnaires indicated that we should have provided more training, for both PBR and error abstraction.
A How did the lack of training effect what you did?

B Was there a steep learning curve with either technique?

C Can you suggest a better way of doing the training?

4 Questionnaires showed that in general people were unhappy with the documents, and thought they were too small, too artificial, had too many defects. In your opinion, would it be worth trying either technique (PBR or error abstraction) on larger, more " real" documents?

A In the context of a classroom assignment, what could we have done to make the documents " more real" ?

5 Are there additional perspectives you think we should add to PBR? (For example, some people suggested a performance or synchronization technique.)

6 In previous experiments with PBR, we used a less formal procedure and people asked for more formality. Do you feel the technique you used had the right level of formality?

7 In terms of the assignment, do you think having the team meetings was a useful experience?

A Did your team do anything to prepare for the meeting? In retrospect, should you have?

B Did your team physically meet, or just work on the list separately?

C Did all the team members participate, or did one member volunteer for or get stuck with a majority of the work?