Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-oriented Programming*

Gail C. Murphy, Robert J. Walker, and Elisa L.A. Baniassad

Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, B.C., Canada V6T 1Z4
{murphy,walker,bani}@cs.ubc.ca

July 24, 1998

UBC Computer Science TR-98-10

Abstract

Two of the most important and most difficult questions one can ask about a new software development technique are whether the technique is useful and whether the technique is usable. Various flavours of empirical study are available to evaluate these questions, including surveys, case studies, and experiments. These different approaches have been used extensively in a number of domains, including management science and human-computer interaction. A growing number of software engineering researchers are using experimental methods to statistically validate hypotheses about relatively mature software development aids. Less guidance is available for a developer of a new and evolving software development technique who is attempting to determine, within some cost bounds, if the technique shows some usefulness. We faced this challenge when assessing a new programming technique called aspect-oriented programming. To assess the technique, we chose to apply both a case study approach and a series of four experiments because we wanted to understand and characterize the kinds of information that each approach might provide when studying a technique that is in its infancy. Our experiences suggest some avenues for further developing empirical methods aimed at evaluating software engineering questions. For instance, guidelines on how different observational techniques can be used as multiple sources of data would be helpful when planning and conducting a case study. For the experimental situation, more guidance is needed on how to balance the precision of measurement with the realism necessary to investigate programming issues. In this paper, we describe and critique the evaluation methods we employed, and discuss the lessons we have learned. These lessons are applicable to researchers attempting to assess other new programming techniques that are in an early stage of development.

*This research was funded by Xerox Corporation, a Canadian NSERC research grant, and a University of British Columbia graduate fellowship.

Keywords: empirical study, software development technique, qualitative assessment, case study, experiment

1 Introduction

Two of the most important and most difficult questions one can ask about a new software development technique are whether the technique is useful and whether the technique is usable. One way to evaluate these questions is to make the technique accessible to the greater community and to see whether the approach sinks or swims. This strategy has many pitfalls: useful techniques that are not yet usable can be lost, and usable techniques that are not particularly useful can inhibit the adoption of other, more powerful techniques. The cost of developing a technique to the point where it can be released to the greater community can also be prohibitive.

An alternate evaluation strategy is to subject the new technique to some form of careful empirical study. Various flavours of empirical study are possible, including surveys, case studies, and experiments [Pfl94, ZW98]. These different approaches have been used extensively in a number of domains, including management sciences (e.g., [All89, TG95]) and human-computer interaction (e.g., [RJ89, Pre94]). Direct application of these methods to studying software engineering questions, however, is often difficult. In human-computer interaction research, for example, the focus is generally on innate physiological and psychological characteristics (e.g., 3D perception); with extensive training, such characteristics can be altered with the result that individuals who have undergone training are considered tainted for the purposes of study. In contrast, most software development aids require some form of training, be it the basic knowledge gained in becoming a software developer or additional, specialized training. There is less experience with empirical techniques in the presence of these kinds of constraints.

A growing number of researchers have been confronting the difficulties and are adopting and applying variations of empirical techniques to assess development aids (e.g., [CSKB+89, PSTV97, SWF+96]). Many of these efforts have focused on the application of experimental methods to statistically validate hypotheses about relatively mature methods and techniques. Less guidance is available for a developer of a new and evolving software development technique attempting to determine—typically within some cost bounds—if the technique shows some usefulness. The developer must attempt to choose between and adapt the various study types available in the absence of any explicit criteria about the benefits, limitations, and cost of each approach for studying software engineering concerns.

We faced these difficulties in trying to assess the usefulness of a new software design and programming technique called aspect-oriented programming [KLM+97] (Section 2). Aspect-oriented programming provides explicit support for expressing separate, cross-cutting programming concerns, such as synchronization or distribution. Using the technique, statements about concerns may be placed in separate modules that are then woven together with the base functionality to form an application. The aspect-oriented approach claims to make it easier to reason about, develop, and maintain certain kinds of application code.

To assess the aspect-oriented approach, we undertook both a three-month case study (Section 3) and a series of four experiments[1] (Section 4). We chose to apply both of

[1]We use the term experiment similar to Basili: "a study undertaken in which the researcher has control

these evaluation approaches because we wanted to understand and characterize the kinds of information that each approach might provide when studying a technique that is in its infancy. Our intent in evaluating aspect-oriented programming was not to be able to *categorically* determine whether the new programming approach could or could not meet all of its claims, but rather to explore whether the approach might be useful, and which parts of the approach might help or hinder various parts of the software development process. Our strategy to put into practice these fuzzy concepts of usefulness and usability was to investigate whether there was any evidence to support subsets of the claims about aspect-oriented programming. Some of our studies attempted to isolate individual claims for investigation, while others considered combinations of the claims.

We based our case study method on the exploratory case study method described by Yin [Yin94]. Our experiences suggest that the general techniques discussed by Yin need to be supplemented by domain-specific techniques. For example, when evaluating a software design and programming approach, it would be helpful to have lists of observational techniques that have been found to be useful for understanding the effects of the new software development approach on the development process. Guidelines on how different observational techniques can be used as multiple sources of data would also help solidify the case study methods used to evaluate new approaches.

We primarily based our experimental methods on the human-computer interaction literature (e.g., [Sch87, RJ89, Pre94, McG95]). This literature has the same roots as the experimental software engineering literature [BSH86, Pfl95c, Pfl95a, Pfl95b]. Employing an experimental approach based on these methods proved more difficult than our adaptation of the case study method. For instance, it was difficult to balance the precision of measurement with the realism necessary to investigate programming issues.

Similar to others [Yin94, Pfl94], we distinguish between case study and experimental approaches based on the degree of control the investigator has over the environment in which the study is being conducted. We consider it a case study method when the investigator has little control over the environment. For instance, in a case study, an investigator may have little input into how participants order or approach tasks, or may have little control over the materials to which participants have access during the study. We consider it an experimental approach when an investigator is able to control many aspects of the environment, such as dictate the use of certain tools.

In this paper, we describe and critique the case study and experimental methods we employed in the assessment of aspect-oriented programming, paying particular attention to the cost of employing the methods (Sections 3 and 4). We also discuss the lessons we have learned from employing these two kinds of empirical study approaches (Section 5). Throughout the paper, we compare and contrast our approaches with the approaches that other researchers have used in studying similar software engineering issues. The contributions of this paper are in the synthesis of our experiences, not in the individual methods we chose to employ.

In short, we found the case study approach a more effective means of assessing the broad usefulness and usability questions of interest for a technique in its early stages of development. Although both case study and experimental methods are costly, early evaluation

over some of the conditions in which the study takes place and control over (some aspects of) the independent variables being studied" [Bas96, p. 444].

of aspect-oriented programming using both empirical approaches has been beneficial; the results of our assessment have been incorporated into further development of the technique. We believe the methods we have developed and the lessons we have learned are applicable to other researchers attempting to assess other new programming techniques.

2 Aspect-oriented Programming

Some design decisions are difficult to express cleanly in code using existing programming techniques. In object-oriented programming, for example, code to support the distribution of the system over multiple machines often ends up spread across multiple classes and methods.

Aspect-oriented programming is a new programming technique intended to enable a more modular expression of these design decisions, referred to as aspects, in the actual code [KLM+97]. As Kiczales and colleagues have noted, one reason aspects have been difficult to capture is that the decisions cross-cut the structure chosen to provide a system's functionality.

To better support the expression of cross-cutting design decisions, aspect-oriented programming uses a component language to describe the basic functionality of the system, and aspect languages to describe the different cross-cutting properties. An aspect weaver is then used to combine the components and the aspects into a system.

Several different aspect-oriented programming systems have been built, including AML, an environment for sparse matrix computation [ILG+97], and RG, an environment for creating image processing systems [MKJ97]. Our empirical studies were performed using the AspectJ™ aspect-oriented programming system [Asp98]. AspectJ uses a slightly modified form of Java™, called JCore, for the component language and supports two aspect languages: COOL for expressing synchronization concerns, and RIDL for expressing distribution concerns. The JCore language removes overloading of methods, the `synchronized` keyword, and the `wait`, `notify`, and `notifyAll` methods from Java to ensure appropriate semantics when the aspect languages are used.

Since we focus in this paper on the methods used in our experiments, we do not describe AspectJ in detail. For the case study and experiments, AspectJ was used from within the Microsoft Visual J++™ environment running on Microsoft NT® workstations. A number of versions of AspectJ were used during the case study; only one was used for the experiments.

With AspectJ, developers write classes in JCore as they would write classes in Java. Synchronization issues can be specified on a per-class or per-object level using COOL: COOL aspects are placed in separate files. RIDL supports the specification of remote interfaces for classes and describes how various objects should be passed across remote interfaces. Similar to COOL, RIDL aspects are placed in separate files.

Figure 2 shows some small snippets of a digital library program written in AspectJ that we used in several of our experiments (Section 4). The code on the left side of Figure 2 is part of a `Query` class written in JCore that represents a query made by a user against one or more libraries. Parts of two methods on `Query` are shown: `addBook` adds a book that is being searched for and has been found into the results list for the query; `numBooks` returns the number of books that have been found. The code on the right side of Figure 2 is part of an aspect written in COOL for the `Query` class. This aspect ensures different threads

JCore

```
public class Query {

  Hashtable books;
  int bookCount = 0;

  public void addBook( Book b,
               Library source ) {
    if( !books.containsKey( b ) ) {
      books.put( b, source );
      bookCount++;
    }
  }

  public long numBooks() {
    return bookCount;
  }
}
```

COOL

```
coordinator Query {
  mutex{ addBook, numBooks };
}
```

Figure 1: Snippets of AspectJ Code.

cannot run the `addBook` and `numBooks` methods concurrently.

The aspect-oriented programming approach claims to make it easier to reason about, develop, and maintain certain kinds of application code [KLM+97]. We conducted a series of empirical studies to begin to evaluate the approach according to some of these claims. When we began these studies, no substantial programs had yet been written using this approach. Furthermore, the AspectJ programming environment, including the aspect languages, were actively evolving.

3 Case Studies

The claims made about aspect-oriented programming are quite broad. Using a case study approach, we were interested in trying to address two narrower, but still broad, questions:

1. Does aspect-oriented programming make it easier to write and change certain kinds of programs?

2. What effect does aspect-oriented programming have on software design activities?

The first question goes to the usefulness of the technique whereas the second question focuses more on usability issues. We were interested in studying these questions by collecting data about multi-person developments using aspect-oriented technology.

To investigate these questions, we undertook two case studies involving a group of four summer interns located at Xerox PARC. One study was composed of two phases: in the first phase, the four interns worked together to develop a distributed game using AspectJ; in the second phase, two of the interns reimplemented the game in Java using a strictly object-oriented approach. In the second study, two interns implemented a distributed library application using AspectJ. The progress of the interns through these projects and their experiences were tracked jointly by researchers at Xerox PARC and at the University of British Columbia (UBC). We describe our case study approach in detail in Section 3.1.

These studies provided some qualitative evidence that the aspect-oriented approach was useful: in a short period of time, the interns, who had little prior concurrent and distributed programming experience were able to produce two complex, albeit small, applications. The studies also provided indications about how the aspect-oriented programming approach can both help and hinder accomplishing a goal. For instance, when an aspect language matched a design concern, such as concurrency, the language provided a vocabulary for expressing and reasoning about that concern. When a particular aspect language is used to try to express a concern not intended by that aspect language, an increase in design complexity can result. In addition, the case studies helped identify:

- a number of challenges possibly facing the usefulness of aspect-oriented programming in other settings,
- a set of concrete AspectJ features that could improve the usability of the approach including the possible addition of aspect languages, and
- a number of potential research directions.

These results are detailed in an internal report [MB97].

Below, we describe the format of our case studies (Section 3.1), analyze the costs of conducting the studies (Section 3.2), and critique the study format (Section 3.3).

3.1 Method

The two case studies were conducted over a two-and-a-half-month period at Xerox PARC from June through August of 1997. Each involved a multi-person development of an application using an aspect-oriented programming environment. Four summer interns took part in the studies: three computer science graduate students and one junior-level computer science undergraduate. Given the infancy of aspect-oriented programming, none of these interns had experience in building aspect-oriented programs. Moreover, although all the interns had knowledge of object-oriented concepts, none of the interns had extensive object-oriented development experience.

To build knowledge of aspect-oriented programming in general, and AspectJ in particular, we first asked the interns to work together for two weeks on several sample problems. We considered this two-week period to be prior to the study period (Figure 3.1).

The first case study was broken into two phases. In the first phase of the study, the interns were asked to consider themselves a small company funded by pseudo-venture capitalists. The company was funded to build several versions of a distributed near-real-time game using AspectJ. The development of this game was broken into three main deliverables. The first deliverable was a version of the game, a space combat game, running for a single user on a single machine where the user played against a computer opponent. The second deliverable was a version of the game running for multiple users on a single machine. The

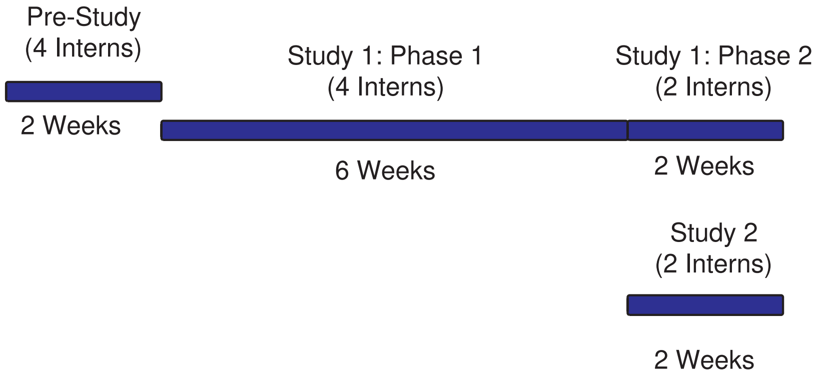

Figure 2: Case Studies Timeline

last deliverable was a version for multiple users running on multiple machines. Each deliverable had an assigned deadline. For each deliverable, the interns were asked to produce a design document and a working version of the system. These deliverables, in particular the design information, were reviewed as part of a regular weekly meeting. The deliverables involved incremental use of different features of the AspectJ environment. This phase of the study took six weeks.

When the game development was completed, two of the interns continued onto the second phase of this study. Over two weeks, the pair built an object-oriented, but not aspect-oriented, version of the game in Java. Originally, we had thought that the four-person game development might take the full eight weeks available for the project: not only was the programming environment untested and evolving; the interns were unfamiliar with each other's design and programming approaches. When time became available, the Xerox and UBC researchers jointly decided to create a second phase to the original study. This second phase provided an opportunity for the students to directly compare development using an object-oriented style with an aspect-oriented style.

The second study was also created as a result of time becoming available in the project. In this study, conducted over the final two weeks of the interns' time at PARC, a pair of interns built a distributed library application using aspect-oriented technology. Although the second phase of the first study and this second study were planned more on-the-fly, we were able to leverage the method that we had been following.

During the two studies, the interns worked in a four-person office area, each with their own NT workstation. As they built the two applications, the interns were asked to reflect on the experience of using aspect-oriented programming, including such issues as what aspect language features were useful, what features were missing, and how aspects affected the organization of the team.

The interns worked alongside the four-person aspect-oriented programming research team at PARC. This research team played several roles during the study period. In the first role, as developers of aspect-oriented technology, the research team provided development support to the interns, responding to problem reports with AspectJ. In the second role, as mentors and supervisors of the interns, the research team set goals for the interns, monitored progress, and evaluated the interns. In the third role, as on-site observers in the study, the team served to both collect and help analyze gathered information on the study.

Two observers from UBC were also involved in the study. These observers participated in the study in four ways:

1. through three on-site meetings held at Xerox PARC comprised of a project initiation meeting in the first week of the study, a mid-term meeting held 4 weeks into the study, and a project wrap-up meeting at the end of the study period;

2. through weekly one- to two-hour video-conference meetings with the interns and members of the research team;

3. through the monitoring of artifacts produced by the interns, most of the artifacts being stored digitally on PARC servers to which the observers had access; and

4. through conference calls to discuss study operations held, at least, weekly with the PARC research team.

We tracked several types of information during the study, including e-mail, hallway conversations, and whiteboard drawings. Table 3.1 provides a complete list of the information tracked.

This data was analyzed by the UBC researchers both throughout the studies and cumulatively, at the end of the studies. We treated the data from both studies together, rather than separately. Although the studies had differences, such as differences in the training levels of the participants, this analysis decision was reasonable given the broad questions of interest. The studies could not be considered as replicas within a multi-case study design because of the rise in expertise with aspect-oriented programming the interns gained during the first study. It was difficult to separate the studies at the end of the period because it was not always possible to determine from discussions in meetings held with the participants—a major data source—which study was the source of any particular comment.

We reviewed the written artifacts, including documents, email, and survey results, attempting to identify and categorize pertinent passages according to a list of keywords we had identified that included such words as "usability", "process", "tools", and "aspect-language". Our intent was to provide an index to the collected data so as to support queries about potentially related evidence. This categorization approach was not successful: we found it impossible to pre-select a reasonably small set of meaningful keywords; adding keywords during categorization required iteration across previously categorized material which was not a reasonable option given the available resources.

Instead, we identified key observations as we reviewed material, tagging the observations with their source. For example, one intern noted in the second survey,

> "I find aspect code to be extremely clear and easy to read. I can only imagine the nightmare of reading through woven Java output looking for deeper meaning behind the slew of Locks and TraversalPatterns."

This quotation became a tagged observation. We then analyzed the observations, drawing together and organizing key observations along development process steps, such as observations about design versus observations about the programming environment. Our analysis also included a structural analysis that compared versions of source code produced based

Information Type	Details and Examples
E-mail	Various mailing lists were used to facilitate communication amongst and between these different groups; all messages were also (manually) logged to a separate file. The information in e-mail included upgrades to the aspect-oriented programming infrastructure, AspectJ feature requests, and general thoughts on aspect-oriented programming.
Weekly Video-conference Meetings	These meetings which involved the interns, researchres, and observors was captured by video-taping the meetings.
Informal Interactions	Substantive hallway and office conversations between interns and the research team was captured by having the researcher log a summary form of the interaction in e-mail.
Documents (Interns)	These documents included documentation on the applications produced and the processes used to produce the applications, proposals of enhancements for AspectJ, and personal journals logging each intern's experience over the study period.
Documents (Researchers)	These documents included AspectJ language specifications and user guides, as well as descriptions of proposed language features and potential directions.
Problem Reports	These reports pertained to problems and fixes for the aspect-oriented programming environment.
Source Code	The source code produced by the interns for the applications.
Whiteboard Drawings	These were drawings sketched by the interns as they worked on the applications. These whiteboard drawings were captured through the use of a zombie board: essentially, a whiteboard with a video camera pointed at it that easily allows sketches to be captured. These captured drawings were stored digitally.
Survey Results (Interns)	Two surveys were completed by the interns about aspect-oriented programming and AspectJ. One survey was distributed at the midpoint of the study period and was discussed by a UBC observer with the interns during an on-site session at PARC. The second survey was distributed at the end of the study period.
Survey Results (Researchers)	The results of a survey distributed to the researchers at the end of the study period about the studies themselves. Only one researcher completed this survey.

Table 1: Information Tracked During Case Studies

Participant Type	Person-Days/ Participants	Number of Participants	Total Person-Days
Intern			
Pre-Study	10	4	40
Study	30	4	120
Researcher			
Preparation Time	5	2	10
Observation Time	3.75	2	7.5
Meeting Time	7	2	14
UBC Observer			
Preparation Time	5	2	10
Observation Time	3.75	2	7.5
Meeting Time	7	2	14
Analysis	7	2	14
Total			237

Table 2: Costs of Case Studies

on diagrams of the "knows-about" relation between classes. The "knows-about" relation was defined by one class naming a second class, either to extend or to access functionality within the second class. This structural analysis provided a means of comparing the complexity of the various versions of the application developed.

3.2 Cost

Conducting the case studies entailed labour costs, equipment costs, and travel costs. We focus here on the labour costs as these costs are the most significant.

As described above, the case studies involved interns, on-site researchers, and off-site observers. Table 3.2 summarizes a low estimate of the hours of involvement of each of these classes of participants. The interns spent approximately two weeks in pre-study activities, and approximately six person-weeks of effort developing applications during the studies. Two of the PARC researchers invested approximately a week of preparation time, approximately five hours per week during the study, and seven days of meetings associated with the studies. The UBC observers invested approximately the same amount of time as the PARC researchers, plus the analysis time consisting of approximately seven days for each observer. The total labour cost of conducting the study was thus in excess of 237 person-days.

3.3 Critique

Empirical social research is commonly evaluated according to four tests [Yin94]: construct validity, internal validity, external validity, and reliability. Construct validity refers to whether appropriate means of measurement for the concept being studied have been chosen; internal validity refers to how a causal relationship is established to argue about a theory from the data; external validity refers to the degree of generalizability of the study; and reliability refers to the degree to which someone analyzing the data would conclude the same results. We first consider how our case studies evaluate against these criteria. Then, we reflect on which aspects of our case study format proved useful, and which aspects of our format did not substantially help generate meaningful results.

3.3.1 Method Evaluation

We designed our case study method based on Yin's exploratory case study model [Yin94]. According to Yin, the main purpose of an exploratory study is to "develop pertinent hypotheses and propositions for further inquiry" [Yin94, p. 5]. Our goal was admittedly broader as we were not only interested in deriving hypotheses based on what we observed, but we were also interested in documenting evidence to support theories about the specific questions of interest. Specifically, we were interested in understanding *how* aspect-oriented programming might help development tasks so as to both begin to assess whether the technique is useful and in which areas further inquiry might be targeted. These questions are common with new software development techniques.

The case study model employed affects the importance to place on the four criteria used to evaluate a case study method. For the kind of exploratory study we undertook, we placed our emphasis on construct validity and reliability over internal and external validity. The reason for this emphasis is that we were more interested in being able to identify believable evidence about the questions underlying the study, then to be able to generalize our theories about aspect-oriented programming.

Our approach to construct validity was to collect data in more than one medium wherever possible. For instance, we asked the interns to document their development processes in the design deliverables they were assigned. We then asked about the processes they used in the weekly video-conference meetings. As another example, usability problems with the programming environment reported during meetings were also later logged to email as problem reports. The use of multiple mediums helped to broaden the data collected by ensuring that the observations of all interns were considered. The use of multiple mediums also helped to corroborate observations: we assigned more weight to an observation that appeared in more than one medium when condensing results from the studies.

The reliability of our study with respect to gathering the same data if the same study had been conducted by others was high. Descriptions of how we gathered data were in place throughout the study. We estimate that the reliability with which other researchers would draw the same results from the data is lower. Reliability in this dimension might have been higher had we determined a more rigorous approach to identifying and classifying observations from the data.

A primary question of interest to software researchers and developers outside Xerox PARC is whether a case study of this nature provides any generalizable results. Some of the results from the study are quite specific to the PARC researchers and their particular instantiation of aspect-oriented programming. Others deal with more general issues regarding the separation of code parts. For instance, are aspects expressed in different aspect languages separate or might they be layered as components are layered?

We believe the insights we gained about aspect-oriented programming have some generalizability because our study participants were representative of many developers, namely they had some but not extensive experience with object-oriented development techniques.

Empirical studies are sometimes criticized for using students as subjects because students are not necessarily representative of practitioners who are typically the target users for a software development aid [FPG94]. We are subject to this same criticism. However, participants must be chosen relative to the claims being investigated and the generalizability desired. When assessment is being performed on a new and evolving technology, students are often accessible and can play a useful, cost-effective role within a study.

Our results may also have some generalizability because we placed the participants into a somewhat realistic scenario where the applications they produced had to be produced according to deadlines. The generalizability of many of the results is dependent on the degree to which the concept of aspect-oriented programming is still evolving.

3.3.2 Useful Techniques

We found the following techniques provided useful data for the case study.

On-site interns as the study participants. Two possible criticisms of this study are that the study participants, the users of the aspect-oriented approach, were co-located with the researchers developing the approach, and that the aspect-oriented programming environment underwent significant change over the course of the study. Since the goals of the study were to broadly understand the issues surrounding aspect-oriented programming, rather than to *definitively* show the value of the approach, these study factors do not undermine the value of the results. Rather, for this kind of study, these factors were an advantage for several reasons. First, since the participants were interns, there was a well-defined period in which the study could and would take place. This time factor limited, in a positive sense, the size of the problems that could be tackled and helped place realistic engineering time pressures on the study participants. Second, the experiences of the interns with the programming environment could be quickly fed back into the research cycle; the research team could use this information to prioritize their support activities.

Outside observers. There were both advantages and disadvantages to the UBC observers not being full-time at the study site. One benefit from being off-site was that these observers could ask the same question multiple times to gauge the similarity of response. The different modes of communication used when off-site, such as e-mail and telephone, seem to make it more acceptable to keep asking the same question. Another benefit was that it was easier for these observers to be objective about the feedback provided by the interns. The major disadvantage from the set-up is that a significant source of information from casual conversations was missed. We attempted to capture some of these conversations by having the PARC researchers log to e-mail conversations deemed as containing interesting information. By far, the most interesting discussions about the development activities were casual conversations we had with the interns as we used the equipment in the "pit" where they worked. These conversations were not captured as we did not find a reasonable way to capture the information without altering the information flow.

In retrospect, it may have been useful to ask the participants to tape their conversations. Of course, this would have entailed an additional serious burden on the investigators in having to transcribe and annotate many hours of audio-tape. The value of this data considering the high cost is questionable.

Deadlines. To help investigate the trade-offs that developers in more realistic settings might have to make when using aspect-oriented programming, we enforced deadlines on the interns. Every week to a week and a half, a deliverable was defined. These deadlines helped to ensure a certain amount of functionality was attempted in the systems under construction. According to the interns, the deadlines were not hard to meet and did not significantly affect their work patterns. Since the deadlines helped to ensure progress was made on the project and did not severely undermine the study, they appear to have been successful.

Video-conferencing. The weekly video-conference held between PARC and UBC was necessary to ensure the off-site observers were up-to-date on the current application developments and to ensure the participants were comfortable with answering questions from the observers. The downside of the video-conferencing was the quality of the PictureTel system used which caused the UBC participants to hear only about 75% of the discussions occurring on the PARC side. Each of these video-conferenced sessions was also video-taped. These video tapes were later analyzed by recording observations during playback; these tapes proved useful as data sources.

3.3.3 Less Useful Techniques

Other techniques did not work as well to provide data for the study.

Journals. We had asked the interns to keep journals, either electronically or within notebooks, to capture their evolving thoughts about aspect-oriented programming. This information did not turn out to be particularly useful because the interns wrote in the journals only periodically over the course of the studies, and because the information was often very general. We did not provide many guidelines for the kind of information to record in the journals and we did not provide any incentives for maintaining this information. Better guidelines or incentives may have lead to more useful information.

Documentation. With each application milestone, we requested documentation on the design, the implementation and the development process used to construct the application. In general, this documentation served mostly to formulate questions to ask the interns. It was much easier to extract the desired information about these development activities through meetings.

Zombie Board. The intent of using the zombie board was to capture those all important sketches that often appear on whiteboards during a development and which can give extensive insight into the design process. Since, in general, the UBC observers had access to the picture without any accompanying explanation, this information did not have sufficient context to be useful. Zombie information might be useful if accompanied by short textual or audio clips describing the information. The zombie board information captured during meetings was useful for later reference.

4 Experiments

The case study method we employed permitted us to investigate broad issues concerning aspect-oriented programming. We were also interested in understanding how aspect-oriented programming eased, or did not ease, particular programming tasks. To investigate three more specific tasks, we designed a set of four experiments:

1. the first experiment compared the ease of *creating* a program using an aspect-oriented approach with an object-oriented approach,

2. another experiment compared the ease of *debugging* in aspect-oriented and object-oriented approaches,

3. a third experiment investigated the ease of *changing* an aspect-oriented program compared to changing a program written in a domain-specific (and object-oriented) language, and

4. the final experiment investigated a combination of these activities.

These experiments were conducted at UBC between September 1997 and May 1998.

In conducting these four experiments, we were constrained by four factors: *the pool of potential participants* available to us was small, *the amount of time* each participant could devote to an experiment was short—especially in comparison to typical development times of even tiny applications, *the cost of running and analyzing the experiments was high*, and since the evaluation of an aspect-oriented approach is complex, *some precision of measurement had to be forfeited* in favour of realism [McG95]. As a result, our "experiments" were set up as semi-controlled empirical studies rather than statistically valid experiments.

As investigators, we had some limited experience running experiments to investigate human-interaction questions, but no experience applying the technique to investigate software engineering issues. We quickly learned some of the differences when conducting software engineering experiments with the result that our first experiment became a pilot study. This change occurred because we ended up refining our experimental method to overcome problems that occurred when conducting the experiment.

We describe the methods we used in the pilot study and the three experiments in Section 4.1. These experiments were successful in gathering qualitative evidence about the usefulness of aspect-oriented programming; most participants that used the approach were enthusiastic about how it supported them in completing the task assigned. In some cases the qualitative evidence was supported by limited quantitative evidence. For instance, in one experiment, we compared the number of times a participant selected a different file to view when debugging aspect-oriented code with the number that occurred when debugging similar object-oriented code.

The experiments also revealed which parts of the approach contribute to its usefulness and usability. For instance, the aspect language used in the debugging experiment operated on methods in component code whereas the participants using the object-oriented approach could operate on statements as well as methods. The granularity limitation of the aspect approach may have contributed to the aspect-oriented users more easily finding and solving certain kinds of concurrency problems. These detailed observations were obtained at less cost than incurred in running the case studies.

Similar to the description of our case studies, we focus below on the format of our experiments (Section 4.1), the costs of conducting the experiments (Section 4.2), and a critique of the approach (Section 4.3). More detail about the experimental setup and results can be found elsewhere [WBM98].

4.1 Method

Table 4.1 provides an overview of the pilot study and experiments. The study and experiments were conducted in the order presented by the table. The pilot study, the debugging experiment, and the change experiment shared a similar experimental method. In Section 4.1.1, we describe the method used for the debugging experiment as representative of these studies. In Section 4.1.2, we describe the method used for the last experiment that encompassed a longer programming activity.

4.1.1 Comparative Experimental Method

We describe the method used in the debugging experiment as representative of our approach. A session for this experiment consisted of the following steps:

1. First, we introduced the participant to the goal, the overall format of the experiment, and showed the participant a running example of the program to be used in the experiment. We then had the participant complete consent forms.

2. The participant was then given thirty minutes to review a series of web pages on synchronization. The participants had been screened (through questioning) about their knowledge of synchronization issues; we wanted to ensure that this information was fresh in their minds and that we were using the same terminology.

3. Next, the participants were asked to review, for thirty minutes, web material we prepared on the programming approach they would be using. Java users were given material on a lock library that they had available for use. AspectJ users were given material on aspect-oriented programming, and the particular aspect language(s) they would be using.

4. The experimenter then walked the participant through the use of the programming environment to ensure the participants could edit, compile, and run the program. Both the Java and the AspectJ participants used the same basic environment, Microsoft's Visual J++ environment. The environment was extended through its standard customization features to incorporate tools for weaving the aspect-oriented programs.

5. The participants were then introduced to a small program in the environment and were given thirty minutes to play with the program.

6. After a break, the participant was given the programming task. The experimenter showed the participant where the program files were located, described the resources available (including the synchronization information, Java books, design documentation on the program, etc.) and gave them a web page describing the bugs they were to find and remove. The participants were asked to "think-aloud" [GKC87] as they

Experiment	Description
Pilot Study	The pilot study investigated the ease of creating an aspect-oriented program. The experiment addressed whether a programmer working with an aspect-oriented language could produce a working multi-threaded program in less time, and with fewer bugs, than a programmer working in an object-oriented language. We selected a small programming problem with concurrency, and had six Java-knowledgeable programmers attempt to produce a solution to the problem: three programmers worked individually in Java, and three worked individually in AspectJ (the JCore component language and COOL aspect language). The running time for an experimental session was three hours for a Java participant and four hours for an AspectJ participant.
Debugging	The intent of this experiment was to learn whether the separation of concerns provided in aspect-oriented programming enhanced a user's ability to find and fix functionality errors (bugs) present in a multi-threaded program. We again compared the performance of participants working with the combination of JCore and COOL in AspectJ, with participants working on the same program written in Java. The participants worked in pairs to find three cascading synchronization bugs we introduced into an approximately 600 line digital library program. Three pairs of participants worked with AspectJ; three with Java. The running time for an experimental session was four hours.
Change	This experiment focused on the ease of changing an existing program. The experiment involved six participants: three working individually in AspectJ using the JCore component language, the COOL synchronization aspect language, and the RIDL distributed aspect language, and three working in the Emerald distributed object-oriented language [BHJL86, RTL+91]. The participants were each asked to add the same functionality into an approximately 1500-line distributed digital library program that they were given (either in AspectJ or Emerald). The running time for an experimental session was four hours.
Combinative	The last experiment involved two participants both working individually in the AspectJ aspect-oriented language. The intent of this experiment was to study a more realistic programming scenario in which a developer was asked to make more substantive changes to a skeleton program than was possible in the earlier experiments. We used participants who had experience building concurrent and distributed systems with existing techniques. The skeleton program on which the tasks were performed was the same problem used in the debugging and change experiments. The running time for a session was eight hours.

Table 3: Experimental Methods Overview

performed the task. The participants were given ninety minutes to perform the tasks. This part of the session was video-taped. The experimenter was present during this session and available to answer questions about the programming environment.

7. At thirty minute intervals, or after each task, such as finding a bug, was completed, the experimenter stopped the participants and asked a series of questions:
   - What have you done up to now?
   - What are you working on?
   - Any significant problems that you have encountered?
   - What is your plan of attack from here on?
8. At the end of the session, which was either when the participants found and removed the three bugs, or the end of the time limit, the experimenter interviewed the participants, amongst other questions asking them to explain their solutions.

The experiment participants were predominantly graduate students, undergraduate students, and faculty in computer science and computer engineering; one participant was from industry. For the debugging experiment which involved pairs, one participant had control of the computer with the programming problem, and the other had access to a report describing the symptoms of the bugs, and on-line documentation. The debugging experiment was the only of our experiments to use this constructive interaction technique [ODR84, Wil95]; in the other experiments, participants worked alone. Participants were remunerated a fixed amount based on maximum time of participation to remove any temptation to take longer to complete the tasks than was necessary.

The number of participants available to us was small: there are not that many programmers and students versed in both Java, and concurrency and distribution issues. Given the small number of participants, we decided to use knowledge we had of the participants' backgrounds to assign the participants to particular parts of the experiments. For instance, we chose pairs for the debugging experiment to ensure that one participant did not dominate the action in any given trial and come to ignore the other participant. Since we were not attempting to determine the usefulness of the technique in arbitrary team situations, this was a reasonable course of action, referred to as blocking [Pfl95c].

Because the number of participants available was small, we had to determine whether or not to reuse participants between trials and between experiments. We decided not to reuse participants in any way so as to avoid biases of experience that would have complicated analysis of the results.

We took an "on-line" approach to analysis. The experimenter(s) actively followed the actions of the participant(s) by listening to what was described aloud and watching their actions via a video monitor displaying the camera's view.[2] Whenever the experimenter lost track of the actions of the participant(s), the question "What's going on?" was asked. We believe this did not happen often enough to be intrusive, but cannot offer good measures for determining how often such querying would be "too often". This approach was not unreasonable given our definition of a realistic environment; in our case, the technique would

[2]This monitor was present in all the experiments, but was not used actively in this fashion during the pilot study.

have been questionable had absolute concentration been required to use it. The experimenter also recorded times of major events and general observations about the progress of participants as they occurred by annotating a copy of the experimental script. We reasoned that if an experimenter could not determine what was going on during the experiment itself, it was unlikely that reviewing the videotape would help. From one questioning period to the next, we forumlated follow-up questions based on observation and attempting to understand the reasoning of the participants' actions. We were careful not to ask questions that could alter the decisions of the participants.

An additional benefit of this on-line analysis approach was that we could detect and correct instances of usability problems or training deficits as they happened, thereby improving construct validity. For example, we decided that, while it was important to require all participants to use the same environment in performing the experimental tasks, we were not interested in how hard or easy it was to use this environment; therefore, assistance was offered whenever any difficulties in using the features of the environment were directly asked about or even noticed by the experimenter. However, questions regarding style or the use of aspect-orientation were strictly out-of-bounds when above the level of "How do I add a file to this project?" or "So if I want to have COOL code for this class, I have to put it in a separate file?". The guideline was that if the question was answered in training, it should be answered by the experimenter. Design questions, such as "Do I need to have a lock here?", were considered out-of-bounds and would have been answered "If it is needed to complete the task"—providing no information.

To give a sense of the events used in our analysis, we describe the three events on which we focused for the debugging experiment: the time, the number of times a participant selected a different file to view, and the instances of semantic analysis performed by the participants that occurred while finding and solving each bug. The time and file switch counts were relatively straightforward to analyze from the video-tape. To quantify the instances of semantic analysis, we reviewed the tapes and recorded the number of times a participant said something to the effect of "let's find out what this does...". We chose to focus on these three events because they provided a basis on which the different programming approaches could be objectively compared.

The data gathered from the videos was also helpful in assessing the qualitative statements made by the participants during the interviews. For instance, one pair using the aspect-oriented approach stated that although the separation was "handy", they were unsure if separation provided an advantage in the end when both the component and aspect code might need to be consulted to solve problems. This pair, however, switched files less in total than any of the Java pairs.

4.1.2 Combinative Experimental Method

Our comparative experimental method provided a means of carefully investigating a particular question by isolating, as much as possible, the tasks to be performed by the participants. In the debugging comparative experiment, for instance, participants were asked to solve, but not identify, problems with an existing program. This task sometimes arises in production software development environments. At least as often, however, developers are required to both find and solve functional problems with their systems. To address the more realistic development situation in which developers must make substantial modifications to a program and make those modifications work, we used a different experimental method that we refer to as a combinative experimental method.

Our combinative experimental method differed from our comparative approach in two ways. First, the tasks assigned to the participants were more extensive. Specifically, participants were asked to sequentially make two modifications to a working program: the first modification was to add support for concurrency into the digital library, the second modification was to add support for distribution into the digital library. These tasks required significantly more design thought and debugging effort than tasks assigned in our comparative experiments. As a result, the running time of an experimental sessions was considerably longer, requiring eight rather than four hours.

The second difference in our method was that we did not run any experimental sessions in a non-aspect-oriented environment. Our rationale for running only aspect-oriented sessions was that we were primarily interested in collecting, through our interviews during a session, qualitative data about the participant's experiences. In particular, we were interested in seeing if the qualitative data we collected from these longer experimental runs was similar to the data collected from the shorter comparative sessions.

Similar to the comparative method, our experimental sessions included some training time. The same training materials were used for both kinds of experiments. Also similar to the comparative method, we video-taped the participants during the session and interviewed them at defined intervals with a pre-set list of questions. Participants were instructed they could record, simply by speaking, any observations of interest as the session progressed: they were not instructed to talk-aloud.

4.2 Cost

Similar to the case studies, we focus on our labour costs involved in experimental preparation, execution, and analysis. Table 4.2 summarizes the cost for each of the four experiments conducted. The values given in Table 4.2 constitute lower-bounds on the actual cost.[3]

The preparation time for each experiment includes time for preparing materials, such as web pages and program skeletons, time for preparing an experimental script and conducting dry-runs, and time for meeting to discuss the experimental format. As some materials were re-used for more than one experiment, we only included their preparation time when they occurred. Costs for executing experiments are for both the combination of the experimenter's and participants time in running experimental sessions.[4] The analysis costs were difficult to gauge but include the time to review video-tapes and compare collected data. Since much of the analysis for the pilot study and the debugging experiment happened together, we placed this analysis value under the debugging experiment. Since much of the analysis for the change and combinative experiment happened together, we placed this analysis value under the combinative experiment.

The overall cost is less for the experiments (145 person-days) than the case studies (237 person-days). However, more of the costs in the case studies relate to time spent with the participants using the technology; more of the time in the experiments was spent in preparation and analysis.

[3]To reviewers: Please note that we are in the progress of completing our analysis on the change and combinative experiments so these cost figures are approximate.

[4]We include both times to be consistent with the values reported in Table 3.2.

Experiment	Preparation	Execution	Analysis	Total
Pilot Study	10	7	12	29
Debugging	37	10	–	47
Change	37	9	–	46
Combinative	4	5	14	23
Total				145

Table 4: Experimental Costs in Person-Days

4.3 Critique

Similar to the case studies, we can evaluate our approach against the four tests of construct validity, internal validity, external validity, and reliability. In this evaluation, we focus on the comparative experiments; we return to the combinative experiment when comparing the case study and experimental approaches (Section 5).

For the comparative experiments, we were more concerned with construct and internal validity than with external validity and reliability. We placed our focus on the former because we wanted to ensure the results were meaningful to the overall question of interest: does aspect-oriented programming show any promise of easing programming tasks? If our analyses had shown that the participants using aspect-oriented programming had taken longer on their tasks or experienced great difficulties, we did not want their difficulties blamed on other factors, such as difficulties with the programming environment.

Our approach to achieving internal validity was to ensure the different groups—aspect-oriented versus non-aspect-oriented—had access to as-similar support as possible, with variances limited as much as possible to the features of interest. For the debugging experiment, this translated to building a pair of synchronization lock classes in Java that were identical in functionality with the woven output from AspectJ source code. The Java participants were provided this library and documentation on its use. This approach allowed the true aspect-oriented properties of COOL, as opposed to its library-like functionality, to be compared with non-aspect-oriented Java code. For the change experiment, we ensured the program structures were as similar as possible between the AspectJ and Emerald versions, varying only when a different structure would be common in one of the languages. One criticism of this approach is that we nudged the participants down a particular path; for instance, participants may have changed the way they would normally have attacked the debugging problem given the Java lock library. This "reduction in realism" was a reasonable price to pay to be able to compare results from the different groups.

As with the case studies, our approach to construct validity was to gather data from multiple sources. For the experiments, one source was the qualitative statements made by the participants during the taped interviews; the other sources were the data analyzed from the tapes. As we noted earlier, sometimes the data from the multiple sources was corroborative, other times it was contradictory. Corroborative data strengthened the result under discussion: contradictory data weakened the result.

Our stress on realism also addressed construct validity. For example, a formal experiment could have been performed to test the efficacy of separation of concerns, one of the properties purported by aspect-oriented programming. But this is only one property of aspect-oriented programming; if we had formally demonstrated the value of separation of concerns, we would not have been much closer to demonstrating the usefulness of aspect-oriented programming but would have done a comparable amount of work.

The reliability of our experiments was high with respect to the procedures we followed in conducting the experiments and analyzing the data. However, as expected, the skills of the participants varied greatly. It was difficult to find participants who met the minimal requirements of our studies, namely experience with Java, concurrent programming, and for the change and combinative experiments, distributed programming. Thus, we did not subject the participants to stringent pre-tests on the scope of their understanding and experience in each of these areas. We do not see any reasonable way we could have further limited the variability in the participants.

The external validity (generalizability) of our experiments was low. In designing the experiments, we knew that our participant pool was limited and had high variability. We also knew that the questions we were interested in investigating were highly dependent on the problems we chose and the environment in which our participants were working. Designing an experiment that could generalize whether or not aspect-oriented programming will allow faster creation of multi-threaded programs, or more efficient debugging, than current techniques is impossible because of these many contributing factors. We wanted to achieve sufficient external validity for our results to have merit with respect to our goals. By using participants with some background in multi-threading and distribution, and by balancing realism in the experiments, we believe we achieved this level.

The experience of designing, conducting, and analyzing this series of experiments identified a set of techniques to us that must be in place for these kinds of studies (Section 4.3.1). We also discuss the techniques we used that were useful (Section 4.3.2). A number of difficulties one may need to overcome are discussed later (Section 5).

4.3.1 Necessary Techniques

Terminology. For these types of experiments, it is essential that all participants be given lengthy exposure to the concepts to be used in the experiment and their mapping, in our case, to the languages of interest. After this exposure they should be tested to ensure they know the information necessary to perform the experiment task. Subtle differences in vocabulary can be problematic for participants to understand the task being assigned. Subtle differences in constructs, such as synchronization constructs, can be a great hindrance to someone attempting to use a language in which they have not frequently programmed synchronization, even if they are otherwise familiar with synchronization concepts.

Participant Training. It is also essential to train participants in the set-up and use of the programming environment. One of the reasons our first experiment became a pilot study was that we showed participants the environment, but did not allow sufficient time for them to interact with the environment prior to the study period. The essential lesson is that it is impossible to test usefulness when usability is at a minimum.

Protocol. To ensure consistency between sessions, the experimenters followed a protocol, consisting of a script of steps to complete and guidelines on interaction with the participants. Scripting operational steps is straightforward; determining appropriate interaction with participants is difficult and delicate. Participants will vary in their understanding, experience and skills; each is likely to ask different questions, requiring different answers.

Since, in our experiments, we were not interested in the efficacy of our training methods, it did not make sense to precisely script the details of communication with participants. On the other hand, not thoroughly scripting interaction can introduce biases into a study when the experimenter casually responds to any query from participants. Instead, we placed well-defined boundaries on what information would be offered to participants and which would not.

4.3.2 Useful Techniques

Timed Interviews. Our original approach to collecting data for the experiments was to have the participants talk-aloud during their session, to video-tape each session, and then later annotate the video-tape. During the pilot study, however, we found that most participants did not provide the information of interest as they talked-aloud. Furthermore, many participants mumbled, since they were essentially talking to themselves, which complicated the annotation process. To get a better sense of how the participants were attacking the given problems, we introduced the protocol of stopping participants every thirty minutes and asking a series of questions. Our initial analysis of the video-tapes then concentrated on these interview segments. These interviews ensured we had participants' views on the questions of interest at different stages and reduced the annotation load.

Constrained Experiment Times. In some cases, particularly during our pilot study, the participants were unable to complete the given task(s) in the time allotted. One way of mitigating this problem is to hold additional dry-run sessions to try and gauge if the experimental time is reasonable. Another approach is to give participants as much time as they need for the task up to some reasonably large maximum, such as several hours. Bowdidge and Griswold used this kind of approach in a study of a program restructuring tool: participants were permitted an additional hour of time after a two-hour defined experimental session time [BG97]. For us, in most cases where time was an issue, it was unlikely that additional time alone would have lead to more consistent (and interpretable) results. Our approach was to constrain the running times to values that seemed reasonable given the dry-runs; this approach reasonably balanced the cost of experiments with the results obtained.

5 Lessons Learned

In this section, we synthesize our experiences and present some of the overall assessment lessons we have learned through our evaluation of aspect-oriented programming. We begin by describing some questions to ask when attempting to select an evaluation method (Section 5.1). Next, we address areas in which particular attention must be paid to maintaining realism (Section 5.2). Finally, we present a synthesis of issues that may arise in designing either a case study-based or experimentally-based empirical evaluation (Section 5.3).

As before, we distinguish between case study and experimental methods based on the degree of control an investigator has over the environment in which the study is conducted. This "definition" introduces a spectrum. The case studies we conducted exerted less control than our combinative experimental method which exerted less control than our comparative experimental methods. The lessons below do not attempt to divide this spectrum on a fine-grained scale as our experiences do not warrant such a detailed treatment.

5.1 Selecting a Method

Suppose you have or are asked to evaluate a new software engineering aid or technology. What method should you choose to conduct your evaluation? Deciding on an appropriate method requires consideration of four questions.

1. What do you want to know about the technology?
2. How stable is the technology?
3. How much are you willing to spend in evaluation?
4. How do you want to use the evaluation results?

5.1.1 Goals of the Evaluation

If it is the broad effects of the new technology that are primarily of interest, we found a case study approach to be effective. With this approach, we were able to gather data from such diverse areas as the design process used by participants to problems with the environment in which the technology was being deployed. Our combinative experiment was an attempt to gather similar, but not quite as broad, qualitative data about multiple facets of tasks in a more controlled setting. We did not find that this evaluation method provided as much data, in part, because the time constraints placed on the tasks restricted the different approaches the participants could try to complete the tasks. We were also able to spend the bulk of the effort involved in conducting the case study on activities involving the use of the technology by the participants. A similar quantity of experience was garnered in the *preparation* for the comparative experiments, but the investigators themselves were effectively the participants in this "unofficial case study". Stressing a technology in different ways by different users is particularly important in the early stages of technology development.

The case study approach was also more effective, for us, in quickly identifying and addressing usability issues with the technology than the experimental approach. A wide range of usability questions and problems surround a new development technology, from the understandability of the error messages or feedback reported by a tool to a user to what arrangement of input to a tool, such as how information is split between aspect and component files, is most effective. Our case study method was sufficiently flexible to allow participants a range of interaction with the technology. The longer duration of the case study also made it possible to try to improve usability problems that arose. In an experimental setting, such flexibility is more difficult to allow because an extra non-random factor would be immediately introduced. Experiments into usefulness, though, cannot ignore usability. In our experiments, we provided immediate feedback about usability difficulties encountered, such as interpreting error messages, to ensure analysis could concentrate on studying the usefulness of the technique. Whether it is reasonable to try to address both usefulness and usability at the same time is an open question. Because usefulness and usability are closely intertwined for new technologies, determining how to investigate them together or how to separate these issues at reasonable cost is an important question.

For both methods, the information that was of the most value was the comparative information. As McGrath points out, in the behavioral and social sciences, the "comparisons to be made are the heart of the research" [McG95]. In the case study and the combinative experiment, we relied on qualitative comparisons the participants made to past experiences. For example, in the combinative experiment, one participant noted:

> I don't think [the aspect language] is as elegant for [distribution] as it was for threads...normally, I just write [classes] and then post-process them to make them network-enabled...remote...like DCOM...so I don't see as big a benefit here [with RIDL] as with threads, but the idea of [the] per-attribute basis is nice.

In the comparative methods, we compared the experiences of participants in the two groups; for instance, considering the number of viewing switches between files that occurred when debugging a problem. To investigate the usefulness of a technique, then, it seems desirable to design a study to make comparisons possible. This can be achieved either in a case study or an experimental format, but is not guaranteed by either approach. For instance, our combinative experiment provided this information largely because we focussed some of our interview questions on the issues, asking the participant to relate to previous experiences.

5.1.2 Stability of the Technology

Selecting a method also requires consideration of the stability of the technology. The greater the control that is desired in a study, typically the greater the investment that is required in preparation time. Both of our comparative and combinative experiments, for instance, required more of the labour cost to be devoted to preparation. This cost may only be reasonable with a stable technology. With a case study, there is often more opportunity to overcome problems that may arise with the technology. For example, the version of the AspectJ programming environment used during the case study changed over the course of the study. It was possible to factor this change in versions into the data analysis given the questions of interest, such as the design processes used. Within an experimental format, however, it was necessary to keep the version of the environment consistent, at least within a given experiment, to permit comparison of results.

5.1.3 Cost of the Evaluation

The cost of evaluation is also an issue, particularly for technologies that are evolving quickly. The predominate cost in the case study method we used was the labour costs of our participants, whereas the predominate cost in the experiments was in the labour required to prepare materials for the experiment. We could have significantly reduced our case study costs by reducing the number of participants. Depending on the technology being studied, this may be a reasonable approach to cost management. It may be more difficult to reduce and manage the costs of experiment preparation.

5.1.4 Use of Evaluation Results

Finally, one must determine the goal of the evaluation results. Experimental methods more rigorous than ours are often advocated [Pfl95c]. These more rigorous approaches have the benefit of striving for statistically valid results that may be more generalizable. Achieving these results typically requires a large number of trials. When studying an immature software development technique that is rapidly changing, the costs incurred in preparing and running experimental trials may not be worthwhile, particularly as the applicability of the results may be limited to a short time span in the evolution of the technique. The difficulty of balancing generalizability of the results with the evolution of the technique is not limited to experiments but applies to case studies as well. Careful design of any study is be necessary to achieve a suitable balance.

5.1.5 Approaching Evaluation

Whatever empirical method is chosen, it is necessary for the investigator to figure out the appropriate balance of construct validity, internal validity, external validity, and reliability. Achieving high levels for all of these factors may not be possible for new technologies. For instance, the hypotheses about the technology may not be sufficiently formed to permit external validity to be achieved. In our experience, none of the methods we used made achieving the desired balance easier than any other.

One way of approaching assessment may be to apply ideas from the spiral model of software development [Boe88]. At the early stages of a technology, assessment effort should concentrate on the broad features of the technology when these are still possible to change. Later, more statistically valid studies can be performed testing hypotheses formed from the earlier exploratory studies. The early studies, though, need not serve solely as hypotheses generators as is sometimes alluded [BSH86]; our aspect-oriented programming efforts show that these early results can provide keys to the usefulness and usability of a technology.

Overall, then, the assessment method to choose should depend on the feasibility of conducting a reasonable study given the cost structure available for the questions of interest.

5.2 Maintaining Realism

Studying questions about how a technology may help the software development process is difficult because it requires maintaining a reasonable degree of realism about the process and factors affecting the process while exerting some control to enable the study. We found there were three areas in which careful attention must be paid to realism: the problem underlying the study, the environment, and the participants. These areas apply to both case studies and experiments.

5.2.1 The Problem

Realism in the problem underlying the study comes in two forms. First, if a limited time is available for the study, as is usually the case, the problem chosen for the participant to tackle in the time available must be representative of a problem arising in a larger software development. Selection of an appropriate problem is particularly difficult in the context of experiments which are typically more time-limited than case studies. For the experiments we ran, selecting appropriate problems that participants could reasonably tackle in the time available was almost impossible since problems involving concurrency and synchronization are generally hard to solve. Our definition of "appropriate problem" was one that was motivating to the participants and reasonably realistic. For the first experiment, we chose a simple version of a non-audio karaoke machine in which text at the bottom of a small window scrolls from right to left and a ball bounces straight up and down above the text. The problem was to synchronize the ball and the text such that the ball would bounce on the start of each word. This problem was motivating, but proved too difficult for the time provided, causing, in part, our experiment to become a pilot study.

The problem must be chosen to ensure that adequate testing of the question of interest is accomplished. For some of our experiments, for instance, we could have chosen to use standard reader-writer problems that are used in many textbooks when discussing concurrency. Using this example, however, might have enabled participants to transfer knowledge they previously had of the problem and solution. By recasting the problem in a different framework, such as the digital library used in three of the experiments, we tried to mitigate this issue of knowledge transfer. The digital library example was more successful for us than the karaoke example; it was still motivating and realistic, but more tractable for the participants to understand. It is difficult to provide general guidelines on how to approach the problem selection problem for experiments beyond suggesting the careful use of dry-runs of your experiment to ensure the problem is tractable.

5.2.2 The Environment

Realism can be introduced into the environment by letting participants work, as much as possible, as developers normally would. For example, the deadlines we set in the case study mimic the real constraint that developers seldom, if ever, can take as much time as required to produce a deliverable. As another example of realism in the environment during the case study, we did not restrict, beyond the programming environment, any tools that the participants wanted to use. A general rule we applied for both the case studies and experiments we conducted was to allow participants to use normal tools and resources except where the use would defeat the purposes of the study.

5.2.3 The Participants

Finally, realism in the participants means picking participants that represent, at least part of, the skill-set of the target users of the software technology. This representation is needed to provide meaningful context for the questions of interest. Since the questions of interest to us involved whether aspect-oriented programming eased development tasks given some programming background on the part of the developer, we could not use any beginning students. This point may seem trivial, but it is an important constraint to recognize, particularly when the participant pool is limited. The number of participants available may affect the kind of study that can be conducted.

5.3 Designing the Empirical Study

Although some guidance is available on the overall design of, most notably, experimental studies for software engineering (e.g., [BSH86, Pfl94]), there is little collected information about two critical pieces of the design: data gathering and data analysis. We spent a significant amount of time trying to design these pieces of our case studies and experiments; our critiques of our methods indicate that they are both areas requiring more attention.

5.3.1 Data Gathering

The fundamental problem we experienced with data gathering was trying to gather data meaningful about a task, such as design or programming, that incorporates so many activities. In other words, achieving construct validity is difficult. Performing these kinds of tasks involves problem solving at both abstract and concrete levels [vMM96], time management, and communicating ideas, among other activities. Previous detailed studies in this general area have used a variety of data sources, including time-diaries in which developers self-report time spent on tasks on a provided form [PSV94], video-tapes of programmers working on assigned tasks [RC96], and structured interviews [CKI88]. Because experimentation in software engineering is relatively immature, little guidance is available about the data sources relevant to activities of interest. A body of knowledge relating researchers' experience with different data sources for investigating different parts of the software development process is needed.

5.3.2 Data Analysis

Ideally, we should have spent additional time determining what data analysis we would perform before conducting our case studies and experiments. This statement is one that is easy to write, but difficult to put into practice because it is not at all evident how we could have determined our analysis strategy more fully at the start. Especially when conducting exploratory studies in which some analysis is required to determine appropriate observations that drive further analysis, such foresight is difficult to achieve. The experiences we have gained in analyzing our case study and experimental data, though, suggest some areas requiring further technique development.

In the context of experiments, it is possible to search for patterns that occur in different sessions as part of the analysis. Bowdidge and Griswold describe an example of this approach [BG97]. Patterns in part helped us to determine items of interest to analyze in the video-tape we collected during our experimental trials. Even though there were some repetitions in our case studies—for example, multiple versions of the game were developed iteratively—it was difficult to find patterns, perhaps because the patterns were spread over a longer time duration. Some means of abstracting the patterns from longer duration observations would be helpful.

Multiple sources of data were useful to us in several ways during our analysis of the case study data. For instance, we assigned more weight to observations that occurred in more than one medium. However, our approach to identifying and matching observations from different sources was ad-hoc. Techniques that would provide more rigour to this analysis would help improve the validity of results from case studies such as the ones we conducted.

6 Summary

Many different validation methods for software engineering questions exist [ZW98]. When attempting to select an evaluation method for a software development aid, an investigator must typically trade-off three factors: validity, realism, and cost. An investigator may have significant flexibility in each of these factors when assessing a mature technology. A costly study that provides a high degree of validity, for instance, may be feasible. On the other hand, an investigator assessing a new technology may face more stringent constraints: the validity of any study, for example, may be limited by the evolution path of the technology.

We faced such constraints when undertaking an assessment of the new and evolving aspect-oriented programming technique. To study aspect-oriented programming, we applied two basic methods: a case study method, and an experimental method. Since the technique under study is in its infancy, our case study and experimental methods were largely exploratory, yielding qualitative insights into aspect-oriented programming and directions for further investigation. Our case study approach provided results about the usefulness of the technique, about challenges facing the usefulness of the technique, about concrete features that could improve the usability of the approach, and about potential research directions. Our experimental approach provided qualitative evidence about the usefulness of the technique and identified more specific parts of the approach that contribute to its usefulness and usability. Overall, we found the case study approach a more effective means of achieving our initial goals of assessing whether and how aspect-oriented programming might ease some development tasks.

This paper makes two contributions. First, we describe our experiences in and analyze the costs of applying several different evaluation methods: case studies, and comparative and combinative, but non-statistically valid, experiments. Our experiences highlight some strengths and weaknesses of the various approaches and outline data gathering and analysis methods that were successful and unsuccessful. Second, our experiences highlight the value possible in various forms of semi-controlled studies. Particularly for new technologies, these studies can help determine if the technique shows promise, and furthermore, can help direct the evolution of a technology to increase its usability and potential for usefulness. The results of these studies may help the adoption of the technology by convincing early adopters that there is sufficient grounds to try the technology in more realistic settings.

Evaluating software engineering questions about a new technology requires significant investigation into evaluative techniques used in similar domains, such as the behavioral and social sciences, as well as creativity in determining how to map those techniques to the domain of interest. An interchange of experiences with the techniques in different circumstances is a necessary first step to improving the evaluative methods we have available for new design, programming, and other similar techniques. We believe the methods we have developed and the lessons we have learned will be helpful to other researchers attempting to assess other new software engineering techniques.

Acknowledgments

We thank the Xerox PARC Embedded Computation Area group—Gregor Kiczales, John Lamping, Crista Lopes, Jean-Marc Longitier, and Venkatesh Chopella—for their comments on and involvement in in the studies, the use of the AspectJ weaver, and the fast responses to solving the few problems with the environment that occurred. We also thank Robert Rekrutiak and Paul Nalos for their work on experiment setup and contributions to experiment design; the interns who participated in the case studies (Mark Marchukov, Beth Seamans, Jared Smith-Mickelson, and Tatiana Shpeisman); our many anonymous experimental participants; and Martin Robillard for helpful comments on a draft of this paper.

References

[All89] S.L. Allen. A scientific methodology for MIS case studies. *MIS Quarterly*, pages 33–50, 1989.

[Asp98] AspectJ web page, 1998. http://www.parc.xerox.com/spl/projects/aop/aspectj.

[Bas96] V.R. Basili. The role of experimentation: Past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*, pages 442–450. IEEE Computer Society, 1996.

[BG97] R.W. Bowdidge and W.G. Griswold. How software tools organize programmer behavior during the task of data encapsulation. *Empirical Software Engineering*, September 1997.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5), May 1988.

[BSH86] V.R. Basili, R.W. Selby, and D.H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[CKI88] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

[CSKB+89] B. Curtis, S.B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D.A. Boehm-Davis. Experimental evaluation of software documentation formats. *Journal of Systems and Software*, 9(2):167–207, February 1989.

[FPG94] N. Fenton, S.L. Pfleeger, and R. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

[GKC87] R. Guindon, H. Krasner, and B. Curtis. Breakdowns and processes during the early activities of software design by professionals. In G.M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical studies of programmers: Second Workshop*, pages 65–82, 1987.

[ILG+97] J. Irwin, J.M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming OS sparse matrix code. In *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments First International Conference (ISCOPE '97)*, pages 249–256. Springer-Verlag, December 1997.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[MB97] G.C. Murphy and E.L.A. Baniassad. Qualitative case study results. UBC-CS-SE-AOP-1, October 1997.

[McG95] J.E. McGrath. Methodology matters: Doing research in the behavioral and social sciences. In R.M. Baecker, J. Grudin, and W.A.S. Buxton, editors, *Readings in Human-Computer Interaction: Toward the Year 2000*, pages 152–169. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1995.

[MKJ97] A. Mendhekar, G. Kiczales, and Lamping. J. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, February 1997.

[ODR84] C. O'Malley, S. Draper, and M. Riley. Constructive interaction: a method for studying user-computer-user interaction. In *Proceedings of First IFIP Conference on Human-Computer Interaction (INTERACT '84)*, volume 2, pages 1–5. Elsevier, 1984.

[Pfl94] S.L. Pfleeger. Design and analysis in software engineering, part 1: The language of case studies and formal experiments. *ACM SIGSOFT Software Engineering Notes*, 19(4):16–20, October 1994.

[Pfl95a] S.L. Pfleeger. Experimental design and analysis in software engineering, part 3: Types of experimental design. *ACM SIGSOFT Software Engineering Notes*, 20(2):14–16, April 1995.

[Pfl95b] S.L. Pfleeger. Experimental design and analysis in software engineering, part 4: Choosing an experimental design. *ACM SIGSOFT Software Engineering Notes*, 20(3):13–15, July 1995.

[Pfl95c] S.L. Pfleeger. Experimental design and analysis in sofware engineering, part 2: How to set up an experiment. *ACM SIGSOFT Software Engineering Notes*, 20(1):22–26, January 1995.

[Pre94] J. Preece. *Human-Computer Interaction*, chapter Part VI, Interaction Design: Evaluation. Addison-Wesley Publishing Co., Wokingham, England, 1994.

[PSTV97] A.A. Porter, H.P. Siy, C.A. Toman, and L.G. 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.

[PSV94] D. Perry, N. Staudenmayer, and L. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):38–45, July 1994.

[RC96] M. Rosson and J.M. Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3):219–253, September 1996.

[RJ89] S. Ravden and G. Johnson. *Evaluating Usability of Human-Computer Interfaces: A Practical Method*. Ellis Hornwood Ltd., Chichester, England, 1989.

[RTL+91] R.K. Raj, E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.

[Sch87] B. Schneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, chapter 10: Iterative Design, Testing, and Evaluation. Addison-Wesley Publishing Co., Reading, MA, 1987.

[SWF+96] M.A.D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H.A. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 31–40. IEEE Computer Society Press, 1996.

[TG95] E.A. Trahan and L.J. Gitman. Bridging the theory-practice gap in corporate finance: a survey of chief financial officers. *Quarterly Review of Economics and Finance*, 35(1):73–88, 1995.

[vMM96] A. von Mayrhauser and A.M. Mans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[WBM98] R.J. Walker, E.L.A. Baniassad, and G.C. Murphy. Assessing aspect-oriented programming and design: Preliminary results. Technical Report TR-98-03, University of British Columbia, Dept. of Computer Science, 1998.

[Wil95] D. Wildman. Getting the most from paired-user testing. *ACM Interactions*, 2(3):21–27, 1995.

[Yin94] R.K. Yin. *Case Study Research: Design and Methods (Second Edition)*. Sage Publications, Thousand Oaks, CA, 1994.

[ZW98] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, May 1998.