EXPLORATORY STUDY OF THE IMPACT OF VALUE AND REFERENCE SEMANTICS ON PROGRAMMING

Neha Khedekar

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF INFORMATION SYSTEMS

Dr. Gregory Kulczycki, Chair
Dr. Ing-Ray Chen
Dr. William Frakes

August 10, 2007
Falls Church, Virginia

Keywords: Value, reference semantics, aliasing, surveys, empirical analysis, Java

EXPLORATORY STUDY OF THE IMPACT OF VALUE AND REFERENCE SEMANTICS ON PROGRAMMING

Neha Khedekar

(ABSTRACT)

In this thesis, we measure the impact of reference semantics on programming and reasoning. We designed a survey to compare how well programmers perform under three different programming paradigms. Two of the paradigms, object-copying and swapping use value semantics, while the third, reference-copying, uses reference semantics. We gave the survey to over 25 people. We recorded number of questions answered correctly in each paradigm and the amount of time it took to answer each question. We looked at the overall results as well as the results within various levels of Java experience. Based on anecdotal evidence from the literature, we expected questions based on value semantics to be easier than questions based on reference semantics. The results did not yield differences that were statistically significant, but they did conform to our general expectations. While further investigation is clearly needed, we believe that this work represents an important first step in the empirical analysis of a topic that has previously only been discussed informally.

Acknowledgements

I wrote this thesis during what seems to be the busiest year of my life (so far). I could not have done all this without the support and help of my family. I have to thank my fiancé Fahad. He gives me the strength and support to do amazing things. I have to thank my parents, my sister Tanvi for believing in me, being with me through thick and thin and praying for me all the way. I also have to thank my friends – Ashwini Raina, Suvelee Sarpotdar, Natasha Moodliar, Preet Dora, Misha Bindra, Rukmini, Vidya Dass, Geetanjali Kothari and all others, for being there when I needed them any day and anytime to take my surveys. These surveys have been the heart of my thesis.

I want to extend my heartfelt thanks to Dr. Gregory Kulczycki - my Thesis Advisor, who has been a tremendous source of inspiration and strength. As a JAVA programming expert, he has helped me to view programming and object oriented methodology from a different perspective and improve my analysis and reasoning skills. His clarity of thought and precision about what was required is something I will always cherish. His inexhaustible and unfailing knowledge on this subject has steered me in the right direction throughout my thesis work. He has remained as cool and patient as possible with me at any given point of time while providing me his unwavering support.

I want to also extend my gratitude to Dr. I. R. Chen and Dr. W. Frakes for being my thesis committee members and for giving me their valuable suggestions as well as co-operating with me all the time.

Finally, I give many thanks to each one who has helped me study at Virginia Tech over this period.

Table of Contents

List of Figures

List of Tables

1 Introduction

It is generally accepted by researchers that programs involving references are more difficult to reason about than those that do not [HLW92]. Both practicing programmers [Koe98] [Cop91] and formalists [Hoa75] [NVP98] [LS99] have acknowledged the problem. Despite this widespread acceptance, researchers have not made any rigorous attempts to quantify the problem, and many questions remain unanswered. For example: Is a program written using value semantics more likely to be correct than a similar program written using reference semantics? Does correct code take less time to write if value semantics are used? Are certain kinds of programs more likely to be affected by the distinction between value and reference semantics than others? Does the particular programming paradigm make a difference – even if the paradigms both use value semantics? These are questions that have not been put to any kind of empirical tests. Without empirical evidence, we cannot even say for certain that reference semantics always complicates understanding. The Java language uses reference semantics for all objects, so an experienced Java programmer may find it easier to think of variables as references to objects rather than directly as objects. It may be the case, therefore, that experienced Java programmers tend to write reference-based code correctly more often than they would write value-based code correctly.

In this thesis, we explore two approaches for measuring the impact of reference semantics on programming and reasoning. In the first approach, we formally specify the behavior of two similar components: one that uses value semantics and one that uses reference semantics. We measure the size of both specifications and count the number of variables that each specification uses.

For the second approach, we designed a survey to compare how well programmers perform under different programming paradigms. Two of the paradigms, object-copying and swapping, use value semantics, while the third, reference-copying, uses reference semantics. The survey questions involve a container data structure – queues of objects. This

enables us to give relatively simple code examples that gave different answers under different programming paradigms. Though further study is needed, we suspect that the results would generalize to other container data structures (such as stacks and lists), but they are probably not indicative of programs as a whole. We gave the survey to over 25 people with various levels of Java programming experience. We looked at two main hypotheses. The first was that survey participants would answer more value-based questions correctly than reference-based questions, and the second was that value-based questions would take less time to answer correctly. The data obtained from the results of the surveys did not verify either of the hypotheses, but it was suggestive that programs involving container data structures may be comparatively easier to understand when they are written using value semantics rather than reference semantics. A larger study would need to be conducted to make any conclusive claims.

This thesis makes the following contributions:

- It demonstrates that a formal reference-based specification for a component is longer and contains more variables than a comparable value-based specification.
- It proposes a survey design that to measure the impact of reference semantics on programming and understanding for collection data structures.
- It illustrates the problems and the setbacks that we encountered when administering the pilot survey and describes how we overcame many of these problems.
- It introduces a program for randomly generating survey questions and indicates how much of the process can be automated.

The remainder of this thesis is organized as follows. Section 2 describes the related work done in the area of value and reference semantics. It discusses both formal and informal reasoning. Section 3 introduces formal specifications for two Queues components – one that based on values and one based on references. It emphasizes the complexity of the reference-based specification. Section 4 describes the general survey design, and gives

examples of the various programming paradigms involved in the survey: copy, swap and reference. It also outlines the survey generation process. Section 5 describes the interface design of the pilot surveys and gives summary statistics of the results gathered from that survey. It discusses the problems that arose with the pilot survey. Finally, it presents feedback and suggestions received from some of the participants of the pilot surveys. Section 6 describes how we redesigned the surveys based on the feedback obtained from the pilot survey participants. It presents the results of the final surveys in several formats. One combines the data and focuses on the differences between the three paradigm categories, another compares the results of the survey based on the different level of java programming experience. Finally, we compare the two different surveys given to the participants. Section 6 is concludes with a report of the feedback from some of the survey participants. Finally, we conclude and present some ideas for future work.

2 Related Work and Motivation

This section discusses existing research on reference semantics and its impact on reasoning. A large body of research indicates that references and their associated aliasing are problems for both practitioners and formalists [HLW92].

2.1 Formal reasoning with references

As object-oriented languages have become more popular, the problems related to references and aliasing have become more prevalent. This is because traditional object-oriented languages such as Java treat variables as references to objects rather than directly as object values. References require programmers to distinguish between references and values of objects and introduce the potential for aliasing. Aliasing among mutable objects causes dependencies between syntactically unrelated expressions and forces programmers to reason globally about their code [HLW92]. Consequently, researchers interested in precise reasoning about program behavior have made various proposals to help software developers avoid aliasing, control aliasing, and manage its presence [Hoa75] [NVP98] [LS99]. These efforts have mainly targeted the formal methods community. In [CF91], Crank and Felleisen compare value and reference semantics with regard to various approaches to parameter passing. They conclude (p.10) that value parameter passing "seems the most attractive choice." This conclusion is consistent with the findings of other researchers that reference semantics complicate reasoning.

2.2 Informal reasoning with references

In [Koe98], Koenig and Moo suggest that beginning students find it easier to reason about value-like classes. Coplien discusses the benefits and drawbacks of value and reference semantics in [Cop91], claiming that software components designed with value semantics are more easily understood.

Though many researchers seem to tacitly accept that reference-semantics are more complex then value semantics, no one has attempted to measure this complexity. In this thesis,

we intend to take the first steps to finding empirical evidence of this unknown quantity.

2.3 Functional Languages

The research in this thesis involves imperative languages, in which variables can have different values depending on the program state. In contrast, functional programs contain no assignment statements, so variables, once given a value, never change. And a function call can have no effect other than to compute its result [Hug90]. This makes the order of execution irrelevant; one can freely replace variables by their values and vice versa. This freedom helps make functional programs more tractable mathematically than their imperative counterparts [Gol96]. As variables in functional languages have the same value in every state, aliasing is not a problem in functional languages, so they always have value semantics. Functional programmers use the fact that these languages simplify reasoning to argue that software developers should make more use of functional languages. In [HJ94], an exploratory study was conducted to demonstrate the value of functional languages in prototyping, and suggest that this is an area where the functional programming community can have a large and effective impact.

3 Formal Specification and Example

This section formally describes the behavior of two Queue components – one that uses reference semantics and one that does not. It shows that the specification for the Queue that uses reference semantics is more complicated than the specification for the Queue that uses value semantics. It does this by providing formal specifications for both the value-based Queue and the reference-based Queue in the same specification language, and showing that the reference-based specification is longer and uses more variables than the value-based specification.

3.1 Two Different Queue Specifications

Figure 3.1 gives the formal specification for a value-based Queue. In this specification, a queue object is modeled such that its abstract value is a string of Objects. Methods have pre-conditions and post-conditions that describe what they do. The constructor indicates that the queue is initialized as an empty string. The enqueue method ensures the new value of the queue (denoted by "this") equals the old value of the queue (denoted by "#this") concatenated with the singleton string containing the old value of x (<#x>). The dequeue method requires that the queue is not empty. It ensures that the old queue equals the returned object (denoted by "result") concatenated with the new queue. Finally, the getCopyOfFirst method requires that the queue is not empty, and it ensures that the new queue is same as the old queue and the returned object has the same value as the first object in the queue.

```
import spec.math.StringTheory;

public interface Queue {
    model Str(Object);

    public Queue( );
        ensures this = empty_string;
```

```
public void enqueue(Object x);
        ensures this = #this o <#x>;

    public Object dequeue( );
        requires this ≠ empty_string;
        ensures #this = <result> o this;

    public Object getCopyOfFirst( );
        requires this ≠ empty_string;
        ensures this = #this and <result> is_prefix_of this;

}
```

Figure 3.1 Value-based specification of a Queue component

Figure 3.2 gives the formal specification for a reference-based queue. In this specification, the queue is modeled as a location instead of a mathematical string of objects. The global contents variable is a mapping from locations to strings of objects. The extra level of indirection is necessary because with reference semantics variables represent references to objects rather than objects themselves. The constructor ensures that the object referenced by the current queue location is empty. It also ensures that no other queue variables are aliased to new object. The updates clause indicates that the global contents variable is updated during a constructor call. The enqueue method ensures that new queue object equals old queue object concatenated with the object being enqueued. It also updates the contents mapping for the current queue location. The dequeue method requires that the queue object is not empty and ensures that the old queue object equals the returned object concatenated with the new queue object. It also updates the contents mapping for the current queue location. The getRefToFirst method returns a reference to the first object in the contents of the current queue object. Though the result is of type Object, it is assumed that a generic object, like the queue presented here, would be modeled as a location.

```
import spec.math.StringTheory;
import lang.base.MemoryManager;

public interface Queue {
```

```
    var contents: Location -> Str(Object);
    model Location;

    public Queue( );
        updates contents;
        ensures contents(this) = empty_string and
            (forall r: Queue, if Queue.num(#r) ≠ Queue.num(#this)
                then r ≠ this and contents(r) = #contents(r));

    public void enqueue(Object x);
        updates contents;
        ensures this = #this and x = #x and
            contents(this) = #contents(this) o <x> and
            (forall r: Queue, if r ≠ this then contents(r) = #contents(r));

    public Object dequeue( );
        updates contents;
        requires contents(this) ≠ empty_string;
        ensures this = #this and
            #contents(this) = <result> o contents(this) and
            (forall r: Queue, if r ≠ this then contents(r) = #contents(r));

    public Object getRefToFirst( );
        requires contents(this) ≠ empty_string;
        ensures this = #this and <result> is_prefix_of contents(this);
}
```

Figure 3.2 Reference-based specification of a Queue component

3.2 Comparison of Specifications

For uniformity, both specifications have been written using the same specification language: Resolve [SAK00].

Count	**Specification 1**	**Specification 2**
keywords	20	42
identifiers	14	39
Other tokens	38	107

Table 3.1 Comparison of Specifications

Table 3.1 illustrates the number of keywords, identifiers and other tokens present in each of the specifications. “Other tokens” include “(“, “)”, “<”, “>”, “;”, “->”, “=”, “≠”. Specification 2 contains the highest number of keywords, identifiers and other tokens out of the two. This comparison in table 3.1 suggests that formal reasoning would take more time for a human and consume more resources in an automated tool.

4 Surveys

In addition to our empirical analysis of value-based and reference-based specifications, we conducted surveys to compare how well programmers performed when they were given questions involving both value-based and reference-based code. The survey design went through two major iterations. The pilot survey was given to three groups of programmers. The results of the pilot surveys were highly inconsistent. We asked the survey participants for feedback on this survey, and we found that they experienced several problems and were confused about some aspects of the survey. As a result of this feedback, we redesigned the survey and distributed it again. The data from the final survey was less erratic and the feedback from the participants was more positive.

4.1 Hypotheses

Our null hypothesis (H0) states that the difficulty in understanding code based on value semantics is no less and no greater than the difficulty in understanding code based on reference semantics. We wanted to see if we could reject this null hypothesis by validating the following specific hypotheses:

1) Participants will correctly answer a greater percentage of value-based questions (those using object-copying and swapping paradigms) they will for reference-based questions (H1).

2) Value-based questions will take less time to answer correctly than the reference-based questions (H2).

Our independent variables include the programming paradigm (copying, swapping, or references) used in the question and the Java experience (novice, intermediate, expert) of the survey participant. Our dependent variables are percentage of questions answered correctly and the time taken to answer each question correctly. This gives us the following formula: (% correct, time for correct answers) = f (paradigm, Java experience).

Using the surveys discussed in the next few sections, we looked for results that supported hypotheses H1 and H2.

4.2 General Survey Design

Each survey was comprised of three questions. For each question, the participant was given a pre-existing program state and a block of six programming statements. The participant was then asked to give the program state after the block of code had been executed. The statements for each question in the survey were similar, but they were based on different coding paradigms: one for copying, one for swapping, and one for references. The copying and the swapping paradigms are both based on value semantics, while the reference paradigm is based on reference semantics. Java programmers are familiar with both the copying paradigm and the reference paradigm. Java uses the copying paradigm for built-in types such as *booleans* and *ints*, while it uses the reference paradigm for objects. We included the swapping paradigm because we wanted to see how programmers would perform with a value-based paradigm that they might not be familiar with.

All of the questions involve queues of objects. We did this so that (1) we could minimize the explanation of how the code worked to the survey participants, and (2) we could have the code complex enough that it could give different answers under different paradigms. We expects the results to be generalizable to other container data structures – such as stacks, lists, sets, and maps – but not to programs as a whole. This is because all container data structures can be exhibit the same type of aliasing: an object outside of a container may be aliased to an object inside a container; two objects inside a container may be aliased; or two containers may be aliased. In general programs, however, there can be portions of code in which no objects are aliased even though the potential for aliasing exists. It is unclear how such code would affect understanding. This is an area for future research.

Regardless of paradigm, the statements in each block of code fall into one of the following six categories:

An enqueue operation

- A dequeue operation
- A "get first" operation
- An assignment operation on queues
- An assignment operation on objects
- A clearing operation on objects

The following subsections describe the various paradigms, show how the statements above are implemented in each paradigm, and give examples of tracing tables for each.

4.2.1 Copy paradigm

The copying paradigm is familiar to nearly all programmers. The assignment operator copies object values, and actual parameters are copied to formal parameters in procedure calls. The copy operator is denoted as ":=" to distinguish it from the reference assignment operator (=) used in the reference-based code. Table 4.1 shows how each of the six statement types are represented in code and how they affect the program state. In the table, *t*, *t1*, and *t2* are arbitrary objects and *q*, *q1*, and *q2* are queue objects.

Operation	**Syntax**	**Effect**
Enqueue	q.enqueue(t);	Copies t's value into the last position of the queue
Dequeue	t := q.dequeue();	Removes the first object from the queue and returns it through t
Get First	t := q.getCopyOfFirst();	Copies the value of the first element in the queue into t
Queue Assignment	q1 := q2;	Copies q2's value into q1
Object Assignment	t1 := t2;	Copies t2's value into t1
Object Clearing	t.clear();	Sets t's value to its default value

Table 4.1 Operations using the copy-based paradigm

To see how these operations work in practice, assume that we are given the following pre-existing state: *q1* = ⟨ A B C ⟩ and *q2* = ⟨ D E F ⟩ and *t1* = G and *t2* = H, where *q1* and *q2* are queues of object and *t1* and *t2* are arbitrary objects. Consider the following copy-based code:

```
q2.enqueue (t1);
q2:= q1;
t2:= q1.dequeue ();
q1:= q2;
t2:= t1;
q1.enqueue (t1);
```

The tracing table in Table 4.2 gives the program state after each statement is executed. The final state (state 6) is the post-state of the code block.

State	Statement	Facts
0		q1 = ⟨A B C ⟩ and q2 = ⟨ D E F ⟩ and t1 = G and t2 = H
	q2.enqueue(t1);	
1		q1 = ⟨ A B C ⟩ and q2 = ⟨ D E F G ⟩ and t1 = G and t2 = H
	q2 := q1;	
2		q1 = ⟨ A B C ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = H
	t2 := q1.dequeue();	
3		q1 = ⟨ B C ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = A
	q1 := q2;	
4		q1 = ⟨ A B C ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = A
	t2 := t1;	
5		q1 = ⟨ A B C ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = G
	q1.enqueue(t1);	
6		q1 = ⟨ A B C G ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = G

Table 4.2 Tracing table for copy-based code

When *q1* is assigned to *q2* in the second statement, the value of the *q1* is copied to *q2*. Thus, in state 2, both *q1* and *q2* have the same object value, but they denote distinct objects. Therefore, the statement *t1*:= *q1.dequeue* () modifies the value of *q1*, but it does not modify the value of *q2*.

In all of our surveys, we ask the participant to give the object value of *q1* at the end of the code block. Furthermore, the last statement in each survey question is always an enqueue of *t1* onto *q1*. This effectively forces the participant to know the value of both *q1* and *t1* just before the last enqueue operation. The value of *q1* in the final state of the copy-based code is ⟨ A B C G ⟩.

4.2.2 Swapping Paradigm

The swapping paradigm may not be familiar to all programmers. We used this paradigm to see how programmers would perform with a value-based paradigm that they might not be used to. The assignment operator swaps object values, and actual parameters are copied to formal parameters in procedure calls. The operator ":=:" is used to denote swapping.

One added complication of the swapping paradigm is that it requires in-out parameter passing rather than just in parameter passing like the copying and reference paradigms [WPH02]. For example, when a variable t is enqueued into a queue q, the value of t after the operation is cleared (it gets an initial value). The reason that the swapping paradigm does not copy the object value of t into the queue – as is done in the copying paradigm – is that deep copying is expensive for objects of arbitrary size. The swapping paradigm is intended to have value semantics while avoiding the inefficiencies of deep copying [Har00]. We thought that the in-out parameter passing of the swapping paradigm could be a potential source of confusion for Java programmers, who are used to parameters being passed in only. Since all the code we use in the surveys has Java-like syntax, we thought it might also be a potential source of confusion for the survey participants as well.

Table 4.3 shows how each of the six statement types are represented in code and how they affect the program state. In the table, *t*, *t1*, and *t2* are arbitrary objects and *q*, *q1*, and *q2* are queue objects.

Operation	**Syntax**	**Effect**
Enqueue	q.enqueue(t);	Copies t's value into the last position of the queue and clears t's value
Dequeue	t := q.dequeue();	Removes the first object from the queue and returns it through t
Get First	t := q.getCopyOfFirst();	Copies the value of the first element in the queue into t
Queue Assignment	q1 :=: q2;	Swaps the values of q1 and q2.
Object Assignment	t1 :=: t2;	Swaps the values of t1 and t2.
Object Clearing	t.clear();	Sets t's value to its default value

Table 4.3 Operations using the swap-based paradigm

To see how the operations for the swapping paradigm work in practice, assume that we are given the following pre-existing state: *q1* = ⟨ A B C ⟩ and *q2* = ⟨ D E F ⟩ and *t1* = G and *t2* = H, where *q1* and *q2* are queues of object and *t1* and *t2* are arbitrary objects. Consider the following copy-based code:

```
q2.enqueue (t1);
q2:=: q1;
t2:= q1.dequeue ();
q1:=: q2;
t2:=: t1;
q1.enqueue (t1);
```

The tracing table in Table 4.4 gives the program state after each statement is executed. The final state is the post-state of the code block.

State	Statement	Facts
0		q1 = ⟨ A B C ⟩ and q2 = ⟨ D E F ⟩ and t1 = G and t2 = H
	q2.enqueue(t1);	
1		q1 = ⟨ A B C ⟩ and q2 = ⟨ D E F G ⟩ and t1 = A and t2 = H
	q2 :=: q1;	
2		q1 = ⟨ D E F G ⟩ and q2 = ⟨ A B C ⟩ and t1 = A and t2 = H
	t2 := q1.dequeue();	
3		q1 = ⟨ E F G ⟩ and q2 = ⟨ A B C ⟩ and t1 = A and t2 = D
	q1 :=: q2;	
4		q1 = ⟨ A B C ⟩ and q2 = ⟨ E F G ⟩ and t1 = A and t2 = D
	t2 :=: t1;	
5		q1 = ⟨ A B C ⟩ and q2 = ⟨ E F G ⟩ and t1 = D and t2 = A
	q1.enqueue(t1);	
6		q1 = ⟨ A B C D ⟩ and q2 = ⟨ E F G ⟩ and t1 = A and t2 = A

Table 4.4 Tracing table for swap-based code

The statement *t1.enqueue (q1)* enqueues *t1* into *q1*. Therefore, in state 1, the value of *t1* is copied into the last position of *q1*. In addition, *t1* itself is cleared, that is, *t1* is set to the default value of A. In the second statement, the value of the *q1* is swapped with *q2*. The statement *t1:= q1.dequeue* () modifies the new swapped value of *q1*, but it does not modify the value of *q2* since they are distinct objects. The value of *q1* in the final state of the swap-based code is ⟨ A B C D⟩. Note that this is a different answer from the one we obtained when using similar operations with the copying paradigm.

4.2.3 Reference Paradigm

The reference paradigm is familiar to all Java programmers. The assignment operator copies object references and assigns them to the formal parameters in procedure calls. The reference operator is denoted as "=", as in Java. In fact, the reference-based code is Java code.

Table 4.5 shows how each of the six statement types are represented in code and how they affect the program state. In the table, *t*, *t1*, and *t2* are arbitrary objects and *q*, *q1*, and *q2* are queue objects.

Operation	**Syntax**	**Effect**
Enqueue	q.enqueue(t);	Assigns t's reference into the last position of the queue
Dequeue	t = q.dequeue();	Removes the first object in the queue and assigns its reference to t
Get First	t = q.getRefToFirst();	Gets the reference to the first object in the queue and assigns it to t
Queue Assignment	q1 = q2;	Assigns q2's reference to q1
Object Assignment	t1 = t2;	Assigns t2's reference to t1
Object Clearing	t.clear();	Sets t's value to its default value

Table 4.5 Operations using the reference-based paradigm

To see how these operations work in practice, assume that we are given the following pre-existing state: *q1* = ⟨ A B C ⟩ and *q2* = ⟨ D E F ⟩ and *t1* = G and *t2* = H, where *q1* and *q2* are queues of object and *t1* and *t2* are arbitrary objects. Consider the following reference-based code:

```
q2.enqueue (t1);
q2 = q1;
t2 = q1.dequeue ();
q1 = q2;
t2 = t1;
q1.enqueue (t1);
```

The tracing table in Table 4.6 gives the program state after each statement is executed. The final state is the post-state of the code block.

State	Statement	Facts
0		q1 = ⟨ A B C ⟩ and q2 = ⟨ D E F ⟩ and t1 = G and t2 = H
	q2.enqueue(t1);	
1		q1 = ⟨ A B C ⟩ and q2 = ⟨ D E F G ⟩ and t1 = G and t2 = H
	q2 = q1;	
2		q1 = ⟨ A B C ⟩ and q2 = ⟨ A B C ⟩ and t1 = G and t2 = H (q1 is aliased to q2)
	t2 = q1.dequeue();	
3		q1 = ⟨ B C ⟩ and q2 = ⟨ B C ⟩ and t1 = G and t2 = A (q1 is aliased to q2)
	q1 = q2;	
4		q1 = ⟨ B C ⟩ and q2 = ⟨ B C ⟩ and t1 = G and t2 = A (q1 is aliased to q2)
	t2 = t1;	
5		q1 = ⟨ B C ⟩ and q2 = ⟨ B C ⟩ and t1 = G and t2 = G (q1 is aliased to q2 and t1 is aliased to t2)
	q1.enqueue(t1);	
6		q1 = ⟨ B C G ⟩ and q2 = ⟨ B C G ⟩ and t1 = G and t2 = G (q1 is aliased to q2 and t1 is aliased to t2)

Table 4.6 Tracing table for reference-based code

When *q1* is assigned to *q2* in the second statement, *q1* and *q2* become aliases. Thus, in state 2, both *q1* and *q2* are referencing the same object. Therefore, the statements "*t1 = q1.dequeue* ()" and "*q1.enqueue* (*t1*)" modify the object values of both *q1* and *q2*. The value of *q1* in the final state of the reference-based code is ⟨ B C G⟩. Note that this answer is different from the answers we obtained when using similar operations with the both the copying and the swapping paradigms.

4.3 Survey Creation

The author wrote a program called "Random Question Generator" that generates the questions for the survey. Recall that the survey consists of three similar questions under different programming paradigms. The Random Question Generator effectively goes through the following process to generate a survey:

1. Randomly generates a sequence of five operations. These operations are selected randomly from a pool of six operations – enqueue, dequeue, get first element, object assignment, queue assignment, and clear. A given operation type may be used more than once in a code block. The variables used in the operations (*q1*, *q2*, *t1*, and *t2*) are also chosen randomly, but they must fit the context of the operation.

2. Appends the operation "enqueue" with parameters *q1* and *t1* to the end of the five operations.

3. Randomly generates a pre-existing programming state for *q1*, *q2*, *t1*, and *t2*. This step only occurs with the pilot survey; the revised survey always starts with the same pre-state.

4. Translates the sequence of six operations into each of three paradigms: copying, swapping, and reference.

5. Executes the six statement program for each paradigm using the generated pre-state.

6. If the question violates certain pre-defined parameters for any of the executions, then the question is discarded and the program starts from the beginning. Typical violations include: a dequeue attempt on an empty queue, an enqueue attempt that puts more than five objects in a queue, an object becoming null, and a sequence whose final state results in an empty queue.

7. If the program does not find any problems, it generates the correct post-state for each question.

8. Randomizes the order in which the questions appear on the survey.

Once the Random Question Generator completed this process, the author manually fed the data – the pre-state, code blocks, and variations of the post-state for *q1* – into the Question Writer surveys. This process was carried out to create a single survey.

5 Pilot Surveys

The pilot surveys were created using Question Writer. The pre-states and the code statements generated by the random question generator program described in Section 4.2 were fed into Question Writer. This section describes our initial intentions for the surveys, and explains the problems we encountered. It describes the interface design of the pilot surveys and gives the results of those surveys. It explains why we found the results suspicious and presents the answers to a feedback form that confirmed our suspicions.

Initially, we had planned to administer the survey to three groups of people. The first group would be from a special topics class (CS 5984 – Spring 2006) taught by Dr. Kulczycki. This group of six students would be given 10 surveys in all. Each week, the students would take one survey. We wanted to see examine the effects of multiple surveys given to participants over a span of time. A big problem arose because the surveys were effectively *voluntary* homework assignments. Students seemed engaged in the first two or three surveys, but they soon lost interest, and we began to see patterns in the data that indicated that the students were no longer taking the surveys seriously. In particular, after the first few surveys, we noticed that many students were only spending a few seconds on each question and getting all the answers wrong.

As a result, Dr. Kulczycki spent a class period discussing the survey in detail with the students. During the class, the students took three surveys and gave comments that indicated there was a fair bit of confusion about the survey design. Since some of these students were confused, we sent feedback forms to other survey participants and found that there were similar problems.

The second group of people that we had intended to administer the survey to were the students of an on-line software engineering class (CS 5704 – MIT – Spring 2006). As there were close to 50 students in the class we had high hopes that even one round of surveys could give us some helpful information.

Unfortunately, events at Virginia Tech hastened the end of the semester that year and the survey was never given.

The third group was made up of the author's own personal contacts. These participants were asked to take the surveys from a website created by the author and located at http://surveyquiz.googlepages.com/surveyquiz_vt. Some of these participants were given a feedback form to fill out. The answers to the questions in the form are given in Section 5.3.

5.1 Interface Design of Pilot Survey

The surveys are published in html and hence can be opened by clicking on a web-link on a webpage in a browser. Figure 5.1 shows the instructions page of a sample survey. This page is the first one to be displayed as soon as the survey begins. It contains the instructions to type in a unique name and programming proficiency of the survey taker. It also contains the web-link to the document that survey takers can refer to for clarification of any concepts or doubts regarding the question paradigms.

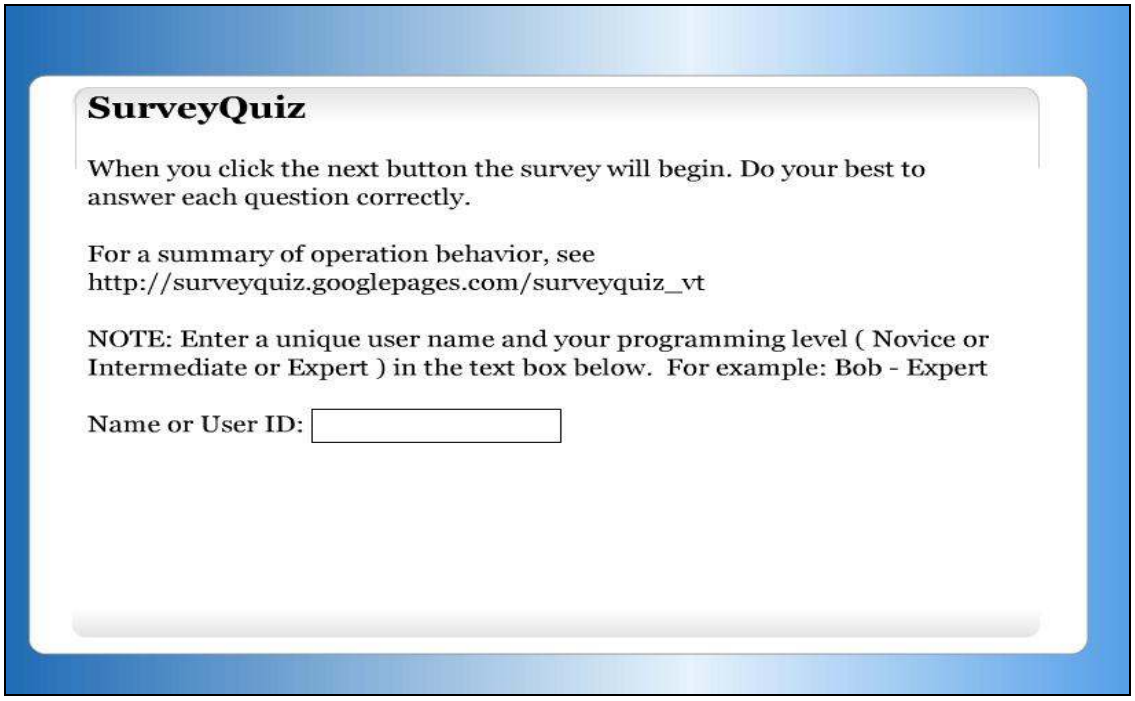

Figure 5.1 Pilot surveys – sample first (instructions) page of the surveys as appeared to survey participants

Figure 5.2 shows a sample swapping-based question on the survey page following the instructions page. It contains the pre-state, the swapping-based code statements and a text field to type in the post-state of the required queue variable.

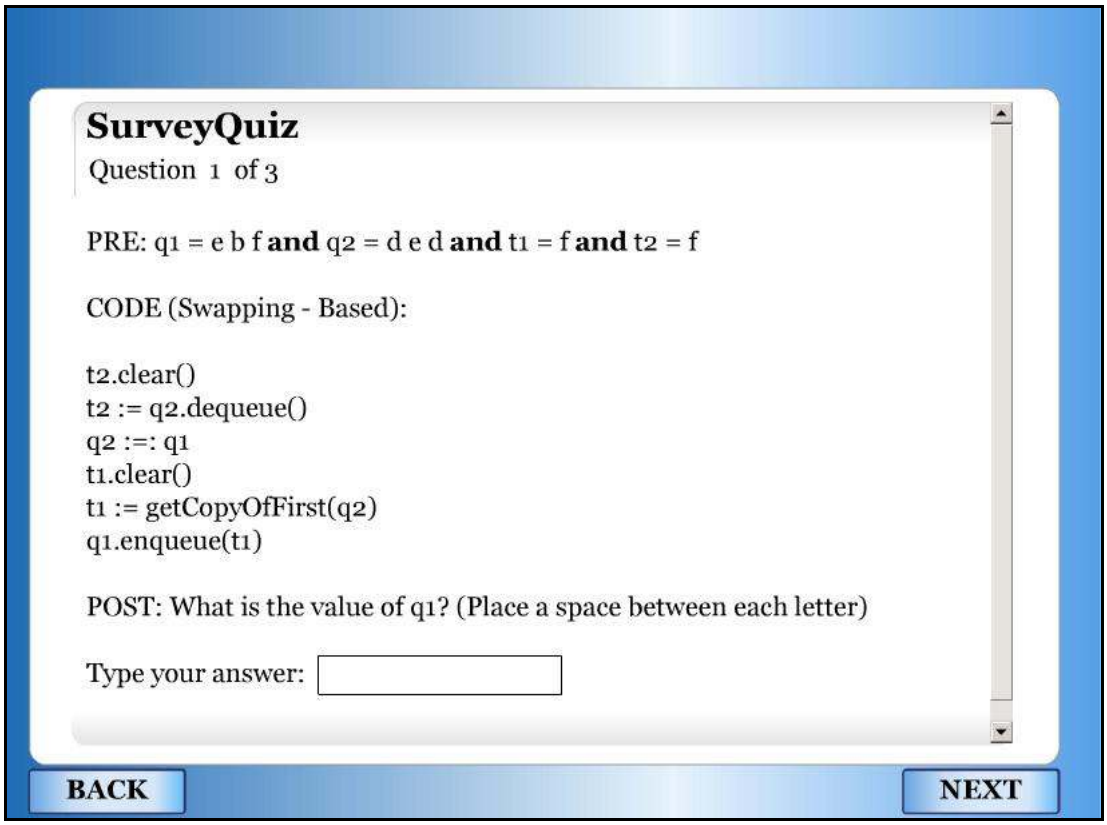

Figure 5.2 Pilot surveys – sample swap question page of the surveys as appeared to survey participants

When all the questions in the quiz were answered, the results were emailed to the author in a report format. The report included the unique name of the participant, the overall score out of the total possible score obtained by the participant, the time spent to complete the survey, the time spent on each question, and the answers given by the participant to each question. A sample report page is given in Figure 5.3.

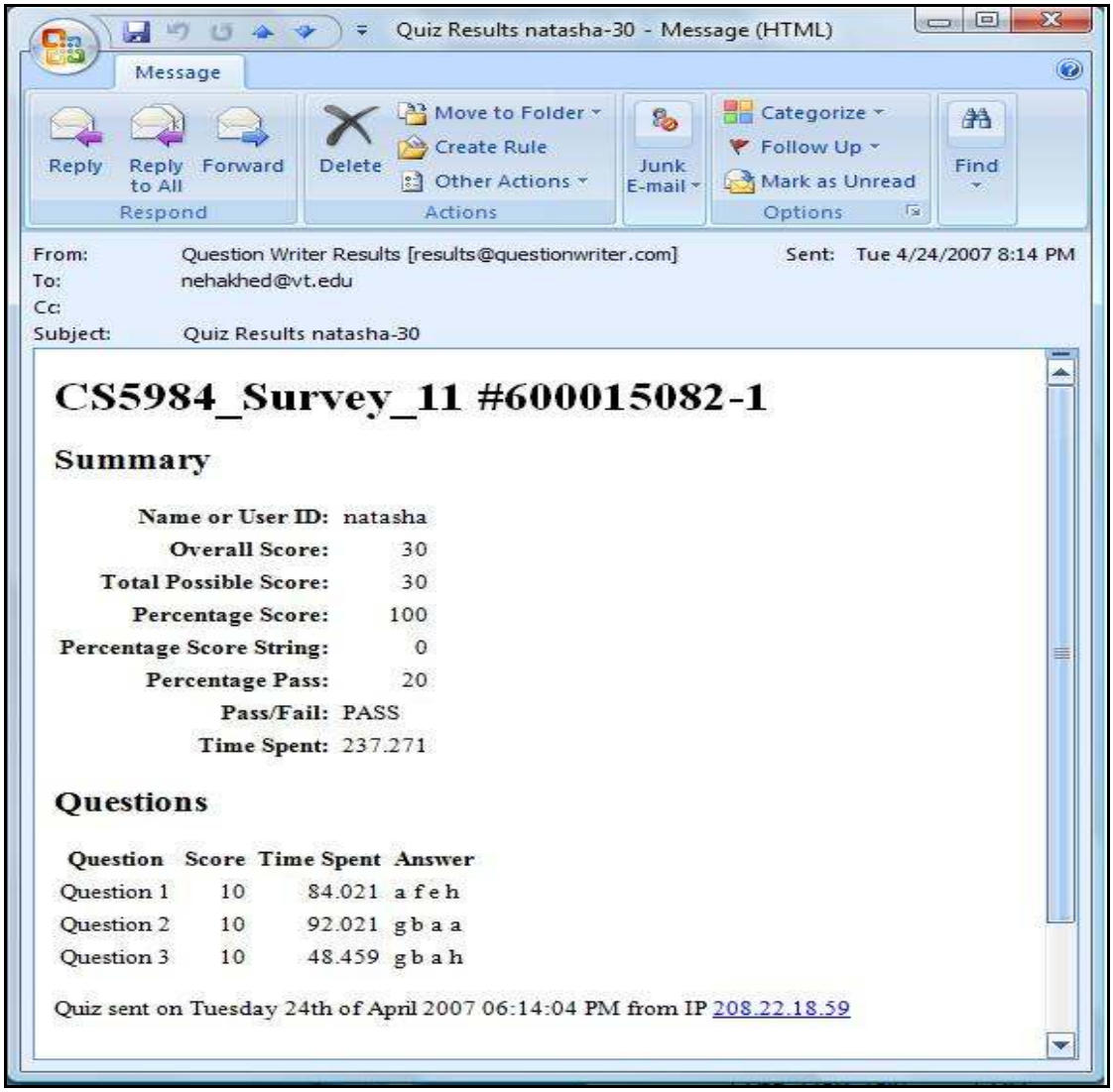

Figure 5.3 Pilot surveys – sample emailed report of the survey results

Figure 5.4 illustrates the PDF document that was included as a reference link on the survey webpage as well as on the survey instructions page so that it could be used as a reference by the survey participants. The participants were asked to review this form before taking the survey. Part of the feedback we received from survey participants was that it was aggravating to have to keep switching back and forth between the form and the survey. The form contains similar information to that given in Tables 4.1, 4.3, and 4.5, but it presents it in the format of pre-state, code, and post-state.

	REFERENCE-BASED	COPY-BASED	SWAPPING-BASED
Tree Assignment	PRE: t1 = g and t2 = h CODE: t1 = t2 POST: t1 = h and t2 = h *(Note: t1 and t2 are aliases)*	PRE: t1 = g and t2 = h CODE: t1 := t2 POST: t1 = h and t2 = h *(Note: t1 and t2 are distinct objects)*	PRE: t1 = g and t2 = h CODE: t1 := t2 POST: t1 = h and t2 = g
Queue Assignment	PRE: q1 = a b c and q2 = d e f CODE: q1 = q2 POST: q1 = d e f and q2 = d e f *(Note: q1 and q2 are aliases)*	PRE: q1 = a b c and q2 = d e f CODE: q1 := q2 POST: q1 = d e f and q2 = d e f *(Note: q1 and q2 are distinct objects)*	PRE: q1 = a b c and q2 = d e f CODE: q1 :=: q2 POST: q1 = d e f and q2 = a b c
Enqueue	PRE: q1 = a b c and t1 = g CODE: q1.enqueue(t1) POST: q1 = a b c g and t1 = g *(Note: t1 aliased to first in q1)*	PRE: q1 = a b c and t1 = g CODE: q1.enqueue(t1) POST: q1 = a b c g and t1 = g *(Note: t1 distinct from first in q1)*	PRE: q1 = a b c and t1 = g CODE: q1.enqueue(t1) POST: q1 = a b c g and t1 = a *(Note: t1 has a distinct initial value)*
Dequeue	PRE: q1 = a b c and t1 = g CODE: t1 = q1.dequeue() POST: q1 = b c and t1 = a	PRE: q1 = a b c and t1 = g CODE: t1 := q1.dequeue() POST: q1 = b c and t1 = a	PRE: q1 = a b c and t1 = g CODE: t1 := q1.dequeue() POST: q1 = b c and t1 = a
Tree Clear	PRE: t1 = g CODE: t1.clear() POST: t1 = a	PRE: t1 = g CODE: t1.clear() POST: t1 = a	PRE: t1 = g CODE: t1.clear() POST: t1 = a
Get First Element	PRE: q1 = a b c and t1 = g CODE: t1 = q1.getRefToFirst() POST: q1 = a b c and t1 = a *(Note: t1 aliased to first in q1)*	PRE: q1 = a b c and t1 = g CODE: t1 := q1.getCopyOfFirst() POST: q1 = a b c and t1 = a *(Note: t1 distinct from first in q1)*	PRE: q1 = a b c and t1 = g CODE: t1 := q1.getCopyOfFirst() POST: q1 = a b c and t1 = a *(Note: t1 distinct from first in q1)*

Figure 5.4 PDF document used as a reference for pilot surveys

5.2 Results from Pilot Survey

The pilot surveys were given to two different groups: the CS 5984 class and the author's personal contacts. The CS 5984 group took two sets of surveys: one in which they completed the surveys as part of voluntary homework assignments, and one in which they completed the surveys during class. Since the surveys they took in class were explained in more detail, the homework results and the in-class results are reported separately. The following sections give summary statistics for all the results of the pilot survey.

5.2.1 CS 5984 Homework Results

These surveys were distributed to the CS 5984 class students. There were five participants and each of them was given 10 surveys. Two of the participants took five surveys each, one of them took six surveys, another took eight surveys and the last one took three surveys. Therefore were total 27 questions that were attempted in each paradigm category.

	Copy	Swap	Reference
Total number of questions	27	27	27
Number correct	7	6	4
Percent correct	25.93%	22.22%	14.81%
Avg. time (all) in seconds	127.60	91.35	72.53
Std. Dev. (all)	278.96	52.87	47.85
Avg. time (correct) in seconds	122.98	136.31	109.25
Std. Dev. (correct)	94.66	71.28	83.49

Table 5.1 Summary statistics of results from CS 5984 homework surveys

Table 5.1 summaries the results from the CS 5984 homework surveys. Copy-based questions resulted in the highest number of correct answers, followed by swapping-based questions. Reference-based questions resulted in the least number of correct answers, with only four correct answers in 27 questions. The average time taken to answer copy-based questions and swapping-based questions is higher than that taken to answer reference-based questions. This is true regardless of the whether the answers were correct or incorrect, and it is a result we did *not* expect. We also did not expect such a low number of correct answers from graduate students who were adept in programming. We suspected that the students might not be taking these voluntary homework assignments very seriously. When we looked at the raw data, we saw that many students had times of only a few seconds on the last few surveys they took. This seemed to confirm our suspicions. Because of this, an in-class survey was given.

5.2.2 CS 5984 In-Class Results

These surveys were distributed to the CS 5984 class students. There were six participants and each of them was given three surveys. Five of the participants attempted all the three surveys each and one of them attempted only two of the three surveys. Therefore, a total of 17 questions were attempted.

Table 5.2 summarizes the results of the CS 5984 in-class surveys. Reference-based questions resulted in the highest number of correct answers, followed by swapping-based and copy-based questions. These results were not consistent with what we expected. However, the differences are small (12, 14, and 15 correct for copy-based, swapping-based, and reference-based paradigms respectively). The average time taken to answer copy-based and swapping-based questions correctly is lower than the time it takes to answer reference-based questions correctly. This result is consistent with what we expected.

	Copy	**Swap**	**Reference**
Total number of questions	17	17	17
Number correct	12	14	15
Percent correct	70.59%	82.35%	88.24%
Avg. time (all) in seconds	103.7595	102.8565	122.6702
Std. Dev. (all)	62.03623	44.04562	49.54841
Avg. time (correct) in seconds	104.6117	106.7644	133.7558
Std. Dev. (correct)	68.62674	46.74498	40.98789

Table 5.2 Summary statistics of results from CS 5984 in-class surveys

No formal feedback forms were given to these students. However, Dr. Kulczycki asked for verbal feedback. He reported that some of the students had real problems understanding the representation of the program state on the surveys. Some thought that all variables were queue variables. For example, they thought that "q1 = a b c" meant that *q1* was a queue of three objects (which is correct) and that "t1 = d" meant that *t1* was a queue of one object (which is not correct). When asked, the students said that the reference-based questions were more difficult than the copy-based or swapping-based questions.

5.2.3 Pilot Survey, Personal Contacts Results

These surveys were distributed to the author's personal contacts. There were seven participants and each of them was given three surveys. Six of the participants completed one survey each and only one of them took all three surveys. Therefore, there were nine questions in all.

Table 5.3 summaries the results from the personal contacts surveys. Reference-based questions resulted in the highest number of correct answers (6), followed by swapping-based and copy-based questions (5). Although there is not much of a difference between the number of questions answered correctly in each category, these results are inconsistent with our expectations. The average time taken to answer reference-based questions correctly is the highest followed by the swapping-based and then the copy-based questions. This *is* consistent with what we had expected.

	Copy	**Swap**	**Reference**
Total number of questions	9	9	9
Number correct	5	5	6
Percent correct	55.56%	55.56%	66.67%
Avg. time (all) in seconds	29.59	124.70	50.99
Std. Dev. (all)	28.09	182.99	41.38
Avg. time (correct) in seconds	36.27	40.37	67.93
Std. Dev. (correct)	27.27	32.51	39.09

Table 5.3 Summary statistics of results from pilot, personal contacts surveys

5.3 Feedback from Pilot Surveys

Given the concerns expressed by the students in the CS 5984 class about the surveys, we decided to elicit feedback from the participants who were personal contacts. We created a feedback form and gave it to four of the six personal contacts who took the surveys. The questions and answers are given below.

Question #1 *How did you come up with answers to each question (using a scratch paper for calculations, computer, or just in the mind)?*

1. Scratch paper for calculations
2. Scratch paper for first operation
3. Using scratch paper and in mind both
4. First one using scratch paper, next two in mind

Question #2 *Which question type was harder to solve and why?*

1. Reference; because it took some time to get acquainted with the operations.
2. Reference; Was unfamiliar with the kind of operations on variables and queue
3. As such the questions weren't hard except there was some confusion in the basic operations of swap, enqueue and dequeue
4. Reference was harder since both the objects need to be kept track of during any operation

Question #3 *Which question type was easiest to solve and why?*

1. Copy-Based operation was easy
2. Copy-Based; Because the answer was obvious especially after solving first question
3. Didn't find any question too easy to solve...all were of same difficulty level
4. Swap easier than copy

Question #4 *Did you have any issues with your computer or other external hindrances/disturbances that hampered your continuity while taking the survey?*

1. No
2. No
3. No
4. No

Question #5 *Did you have any confusion about the survey itself?*

1. No
2. No
3. The purpose of the survey was not clear.
4. It was not clear why the default Tree was "A" and why it was assigned to the tree after the enqueue operation

Question #6 *Did you find any ambiguity in the survey questions or concepts?*

1. No, though operations were little bit different than "Procedural programming language" like C.
2. Yes but only at initial stage
3. Yes there was confusion in the survey since the solved examples PDF stated complete opposite steps for swap, enqueue and dequeue in the queue. Also q1, q2 were tree objects, wasn't clear till the end
4. No

Question #7 *Were you aware that the survey was timed and did the time factor influence your performance?*

1. No. Actually I started the quiz and referred the sample questions simultaneously. So it took me some more time.
2. No. It didn't affect my performance
3. The survey was timed and it would affect the performance wasn't mentioned anywhere
4. Was aware of the timed survey, but solved the questions as per convenience

Question #8 *What is your expertise level in java (novice, intermediate, expert)?*

1. Novice
2. Intermediate
3. Novice
4. Intermediate

Question #9 *When you solved the first question in the survey, did you easily anticipate the answers to other 2 questions without actually solving them?*

1. Yes
2. Yes
3. No
4. No

Question #10 *Did the code statement sequence or difficulty of the operations to be performed in each question affect your performance?*

1. No
2. No
3. No it didn't affect the performance other than the confusion stated in 6
4. No

Question #11 *What preparation was done by you before taking the survey (reading material, setup and so on)?*

1. Just had an overview of the operations
2. Went through the help document
3. The examples and introduction on the webpage was read before giving the survey
4. Read the "Summary of Operation Behavior"

Question #12 *How would you rate your performance in the surveys on the scale of 1 to 5 (5 being the highest)?*

1. 5
2. 5
3. 2.5
4. 4.8

Question #13 *Would you suggest any improvements in the presentation or contents of the survey?*

1. Operations can also be represented by some kind of Animation. It will be easy to understand.
2. Content was good. Reading "Summary of operations behavior" proved to be helpful
3. The confusion stated in 6 should be cleared. Whether the object is pushed in queue from left or right should be made a little clearer. Also the purpose of the survey can be explained a little
4. Nothing

Most participants started out with scratch paper and then solved the remaining questions in their head. This could indicate that some sort of bias might be associated with the first question, and would need further investigation. Some of the participants were confused about what the variables signified and how particular operations were executed. Some participants suggested making use of animation or graphics to illustrate the examples. Most of them found the external PDF document very helpful before they took the surveys, but then they also suggested eliminating the need to go back and forth reading the material before or while taking the survey. Based on this feedback, we decided to revise the survey and administer it again to a larger group.

6 Final Surveys

This section discusses the interface and presents results for the survey after it was redesigned based on the feedback we collected from the participants of the pilot survey. Section 6.1 discusses the how we redesigned the interface based on the pilot survey feedback. Section 6.2 presents the general results of the survey, which was taken by 22 people, including some of the same people who participated in the pilot survey. Section 6.3 looks at the results based on the participants' experience in Java. Twenty of the 22 survey participants took two different surveys. Section 6.4 compares the results from the different surveys from those 20 people. Section 6.5 presents the results of a similar feedback form, which was given to six participants of the final survey. Four of the people who filled out the feedback forms for the pilot survey also filled out feedback forms for the final survey.

6.1 Interface Design of Final Survey

Based on the feedback from the pilot surveys, several changes were made in the design of the survey's interface. These included eliminating the external PDF document, clarifying the abstract representation of variables, and emphasizing different information on the survey page using organization, color and font size.

6.1.1 Elimination of external PDF document

We eliminated the need for survey participants to review information before taking the survey. In the pilot surveys, some participants found it annoying and too time-consuming to have to review external material before even taking the surveys. Also, some participants found it distracting to have to refer to an external document while taking the survey.

We addressed these issues by (1) adding general information on each question page relating to the particular paradigm, and (2) adding an image icon that could be zoomed over, which gave detailed information and examples on the particular paradigm.

Figure 6.1 shows the initial page for the final survey. Note that the survey participants are no longer asked to read any external documents before taking the survey. They can directly begin the survey after entering a unique name in the text field.

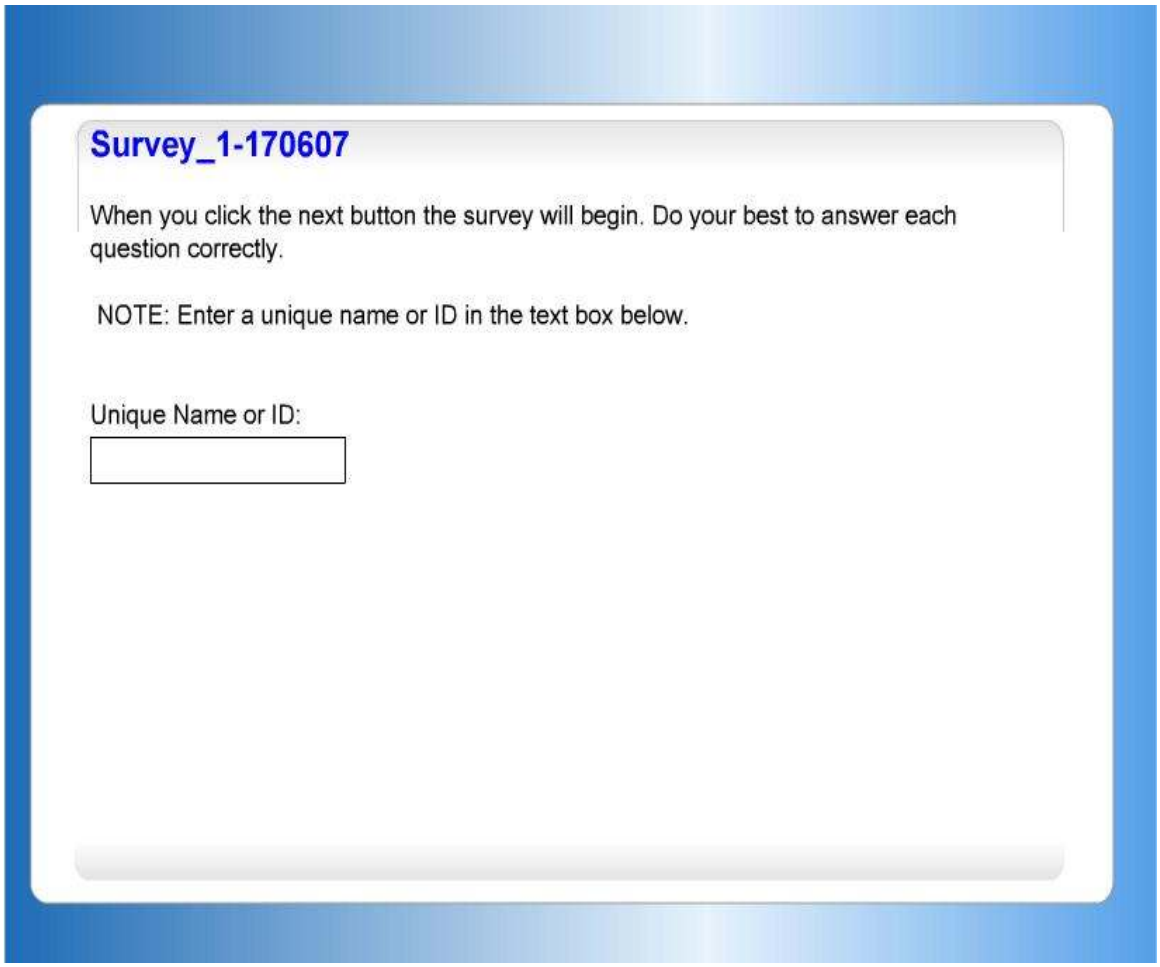

Figure 6.1 The initial page from final survey

Figure 6.2 shows a sample swap question page that contains general information for the swap paradigm. This information is given in blue at the top of the question page. It tells what the swap operator does, describes what the enqueue operation does, tells what the variables represent, and indicates what the initial value is for an arbitrary object. Question pages for the copy paradigm and the reference paradigm have similar information.

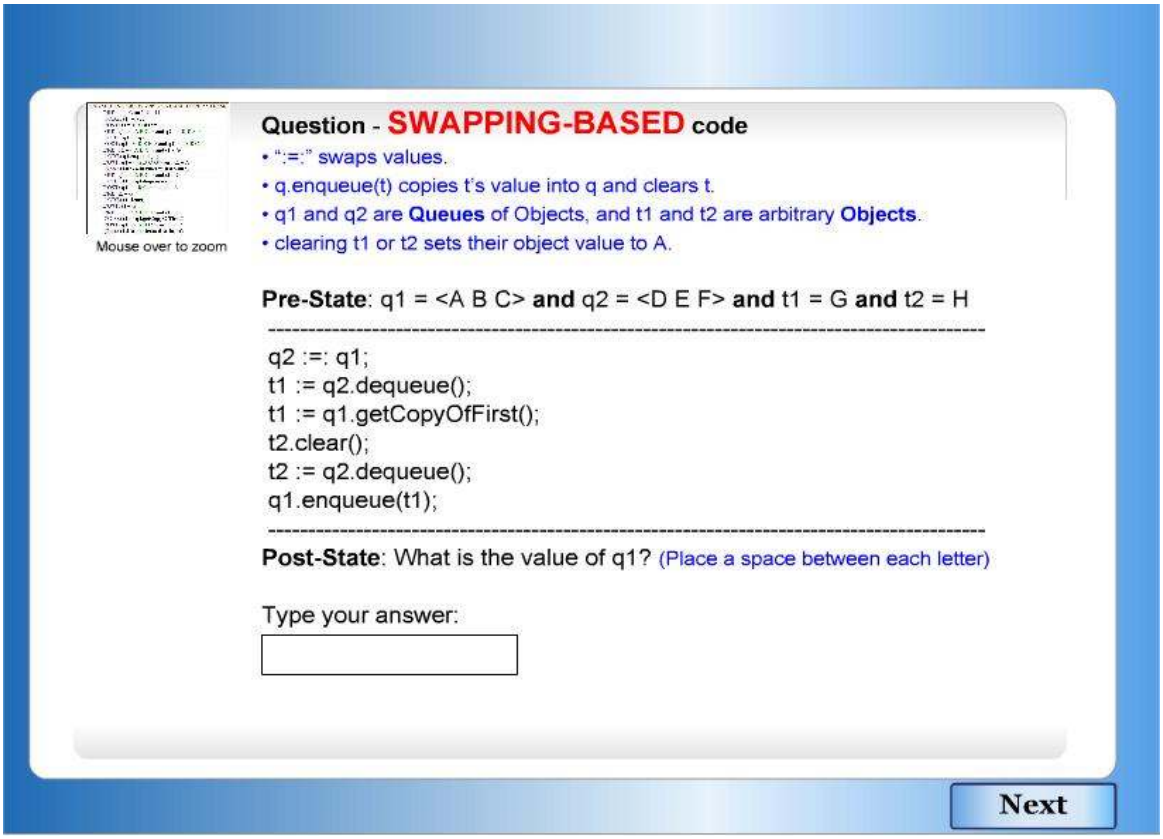

Figure 6.2 A sample question using swapping-based code in the final survey

Figure 6.3 illustrates the detailed information for the swapping-based paradigm. An image icon is located on the top left-hand corner of the each question page. Placing the mouse over that icon magnifies the image, which shows the detailed information for the corresponding paradigm. The detailed information is similar to the information in the external PDF document in the pilot survey. However, a participant in the final survey is able to refer to the detailed information without leaving the question page.

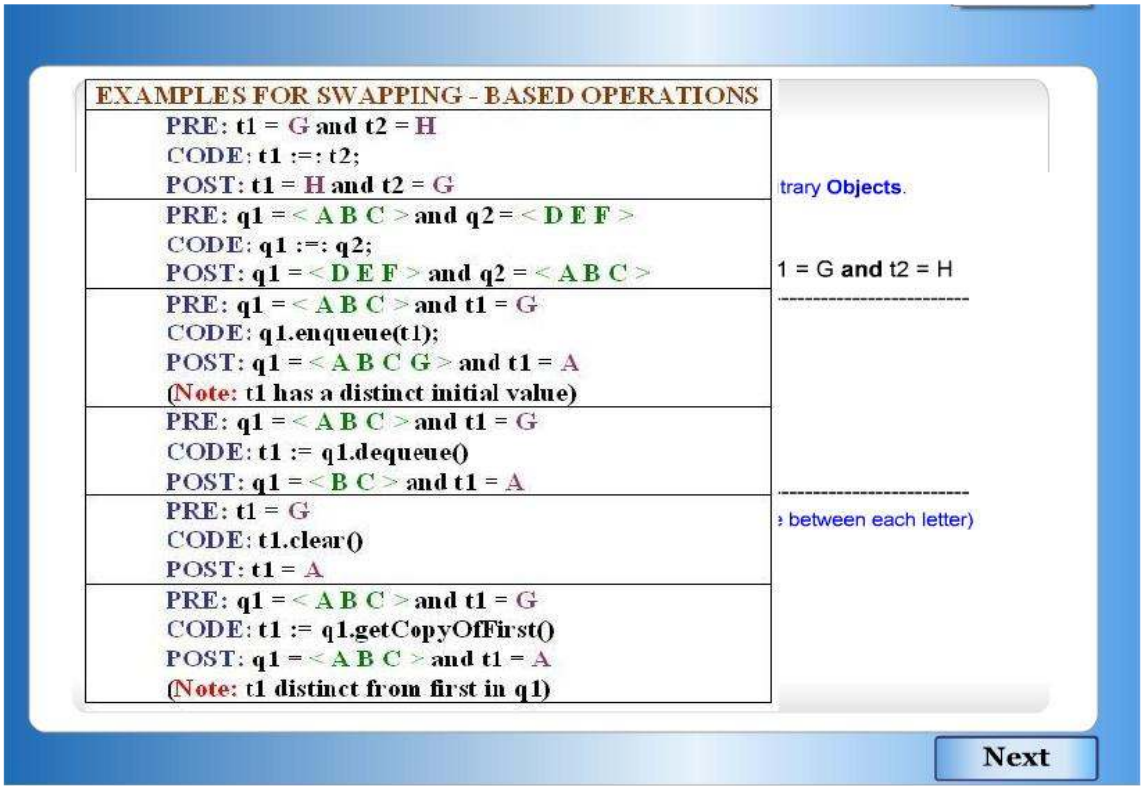

Figure 6.3 Swapping-based question on final survey with information icon fully enlarged

Figures 6.4 through 6.6 show the detailed information for each paradigm in the survey. This information is displayed in the form of an image that can be enlarged by the moving the mouse pointer on it. Each image includes a table showing an example of each type of operation for the corresponding paradigm. Each example lists the pre-state of the variables, the executable code, and the post-state of the variables after executing the code.

EXAMPLES FOR REFERENCE - BASED OPERATIONS
PRE: t1 = G and t2 = H CODE: t1 = t2; POST: t1 = H and t2 = H (Note: t1 and t2 are aliases)
PRE: q1 = < A B C > and q2 = < D E F > CODE: q1 = q2; POST: q1 = < D E F > and q2 = < D E F > (Note: q1 and q2 are aliases)
PRE: q1 = < A B C > and t1 = G CODE: q1.enqueue(t1); POST: q1 = < A B C G > and t1 = G (Note: t1 aliased to first in q1)
PRE: q1 = < A B C > and t1 = G CODE: t1 = q1.dequeue() POST: q1 = < B C > and t1 = A
PRE: t1 = G CODE: t1.clear() POST: t1 = A
PRE: q1 = < A B C > and t1 = G CODE: t1 = q1.getRefToFirst() POST: q1 = < A B C > and t1 = A (Note: t1 aliased to first in q1)

Figure 6.4 Detailed information for reference paradigm

EXAMPLES FOR COPY - BASED OPERATIONS
PRE: t1 = G and t2 = H CODE: t1 := t2; POST: t1 = H and t2 = H (Note: t1 and t2 are distinct objects)
PRE: q1 = < A B C > and q2 = < D E F > CODE: q1 := q2; POST: q1 = < D E F > and q2 = < D E F > (Note: q1 and q2 are distinct objects)
PRE: q1 = < A B C > and t1 = G CODE: q1.enqueue(t1); POST: q1 = < A B C G > and t1 = G (Note: t1 distinct from first in q1)
PRE: q1 = < A B C > and t1 = G CODE: t1 := q1.dequeue() POST: q1 = < B C > and t1 = A
PRE: t1 = G CODE: t1.clear() POST: t1 = A
PRE: q1 = < A B C > and t1 = G CODE: t1 := q1.getCopyOfFirst() POST: q1 = < A B C > and t1 = A (Note: t1 distinct from first in q1)

Figure 6.5 Detailed information for copy paradigm

EXAMPLES FOR SWAPPING - BASED OPERATIONS
PRE: t1 = G and t2 = H CODE: t1 :=: t2; POST: t1 = H and t2 = G
PRE: q1 = < A B C > and q2 = < D E F > CODE: q1 :=: q2; POST: q1 = < D E F > and q2 = < A B C >
PRE: q1 = < A B C > and t1 = G CODE: q1.enqueue(t1); POST: q1 = < A B C G > and t1 = A (Note: t1 has a distinct initial value)
PRE: q1 = < A B C > and t1 = G CODE: t1 := q1.dequeue() POST: q1 = < B C > and t1 = A
PRE: t1 = G CODE: t1.clear() POST: t1 = A
PRE: q1 = < A B C > and t1 = G CODE: t1 := q1.getCopyOfFirst() POST: q1 = < A B C > and t1 = A (Note: t1 distinct from first in q1)

Figure 6.6 Detailed information for swapping paradigm

6.1.2 Clarify the abstract representations of variables in the survey

The feedback obtained from some of the participants of the pilot surveys indicated that the abstract representation of the variables in the code statements was not always intuitive or explicit enough. For example, a queue in the pilot survey might be represented as *q1* = a b c and a tree might be represented as *t1* = d. A number of participants found this confusing since *t1* and *q1* could be mistaken for the same data structure with similar contents but different variable names and sizes. Some participants thought at first that both *q1* and *t1* were queues. To eliminate this confusion, we represented the same variables in the final survey as *q1* = < A B C > and *t1* = D. The brackets were used for queue representations to indicate that they are containers that hold a sequence of objects. No brackets were used for arbitrary objects. The notion of "trees" from the pilot survey was dispensed with in favor of arbitrary objects. In addition, upper case letters were used to indicate object values while lower case letters were used for the variables themselves.

Figure 6.7 shows the revised pre-state of the variables used in a sample question page with copy-based code.

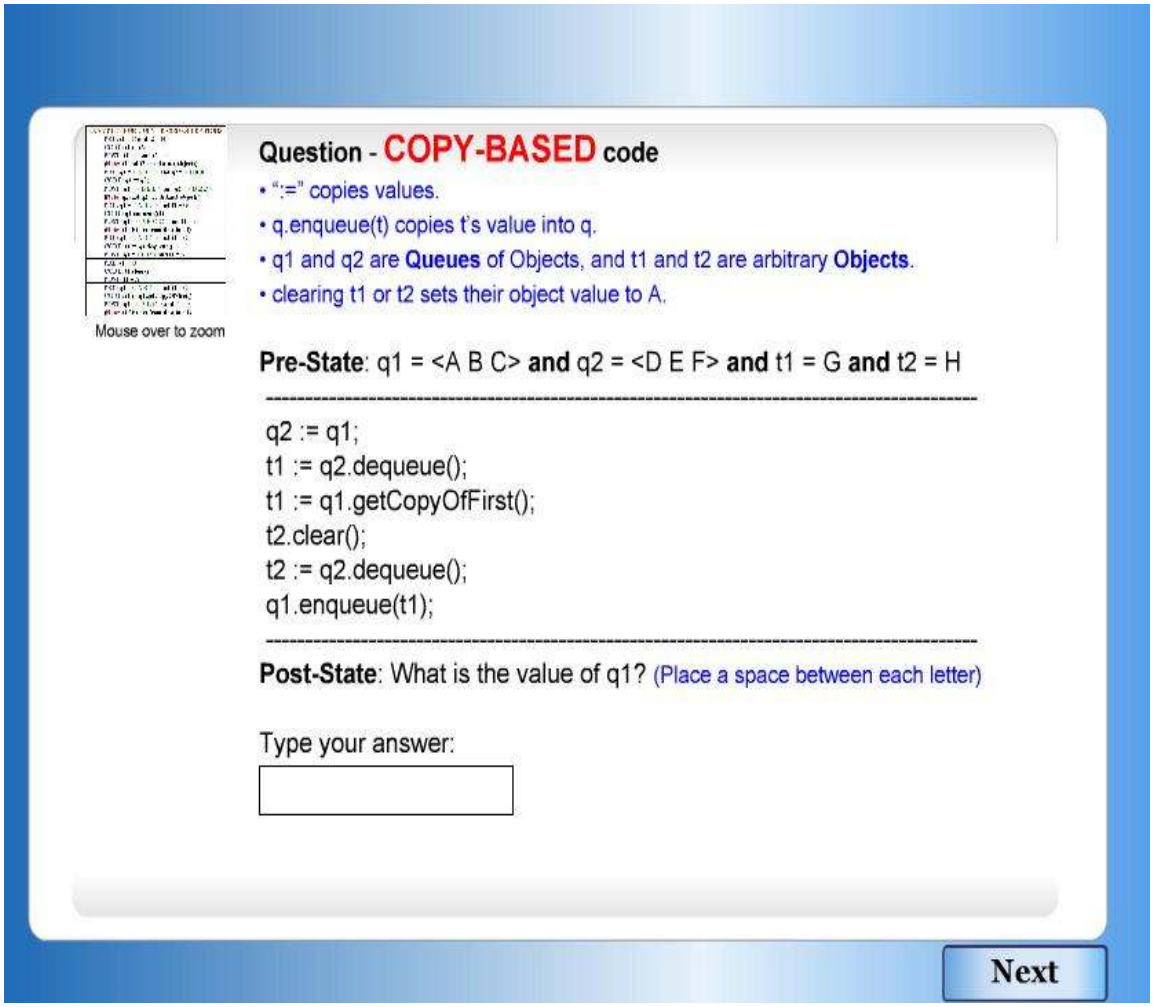

Figure 6.7 Survey page with copy-based code

6.1.3 Other interface redesign issues

Figure 6.8 illustrates the interface of a sample swap question in the pilot surveys. As can be seen, the pre-state, code statements and the post-state sections are listed in a simple manner. There is no highlighting of important points or instructions given. The values of the variables in the pre-state are specified in lower case letters. Also, the pre-state was not constant for every question; rather it was randomly generated for every question and every survey.

To simplify the survey, we kept the pre-state constant for all the redesigned questions, and placed the objects in alphabetical order inside the queues. The general look of the final surveys has significant changes, as can be seen in Figure 6.9. The pre-state, code and post-state sections on the question page are delimited by lines. We eliminated the functionality of going back in the surveys to the previous questions by disabling the "BACK" button. This was done so that the participants could be prevented from going to the previous question and modifying the answers already given.

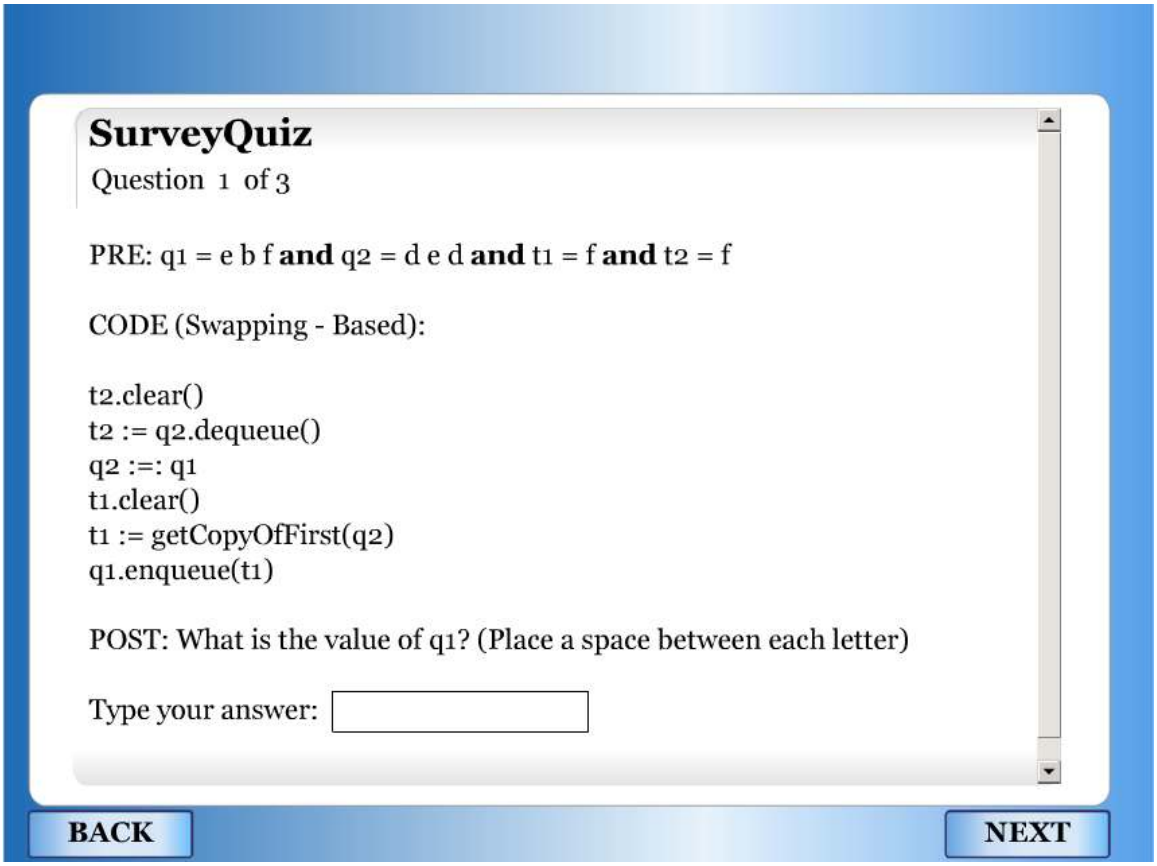

Figure 6.8 Sample swapping-based question page in the pilot surveys

We tried to use colors more effectively in the final survey to make an explicit distinction between the different kinds of information represented. For example, the general information related to a question's paradigm and other survey instructions were displayed in blue, the paradigm (copy-based, swapping-based, or reference-based) was indicated in red, and the rest of the text was black.

The paradigm type was also enlarged and made boldface to give it more emphasis. All of these modifications in the format of the surveys were intended to respond to the suggestions in the feedback given by the participants of the pilot surveys and improve readability and presentation of the surveys.

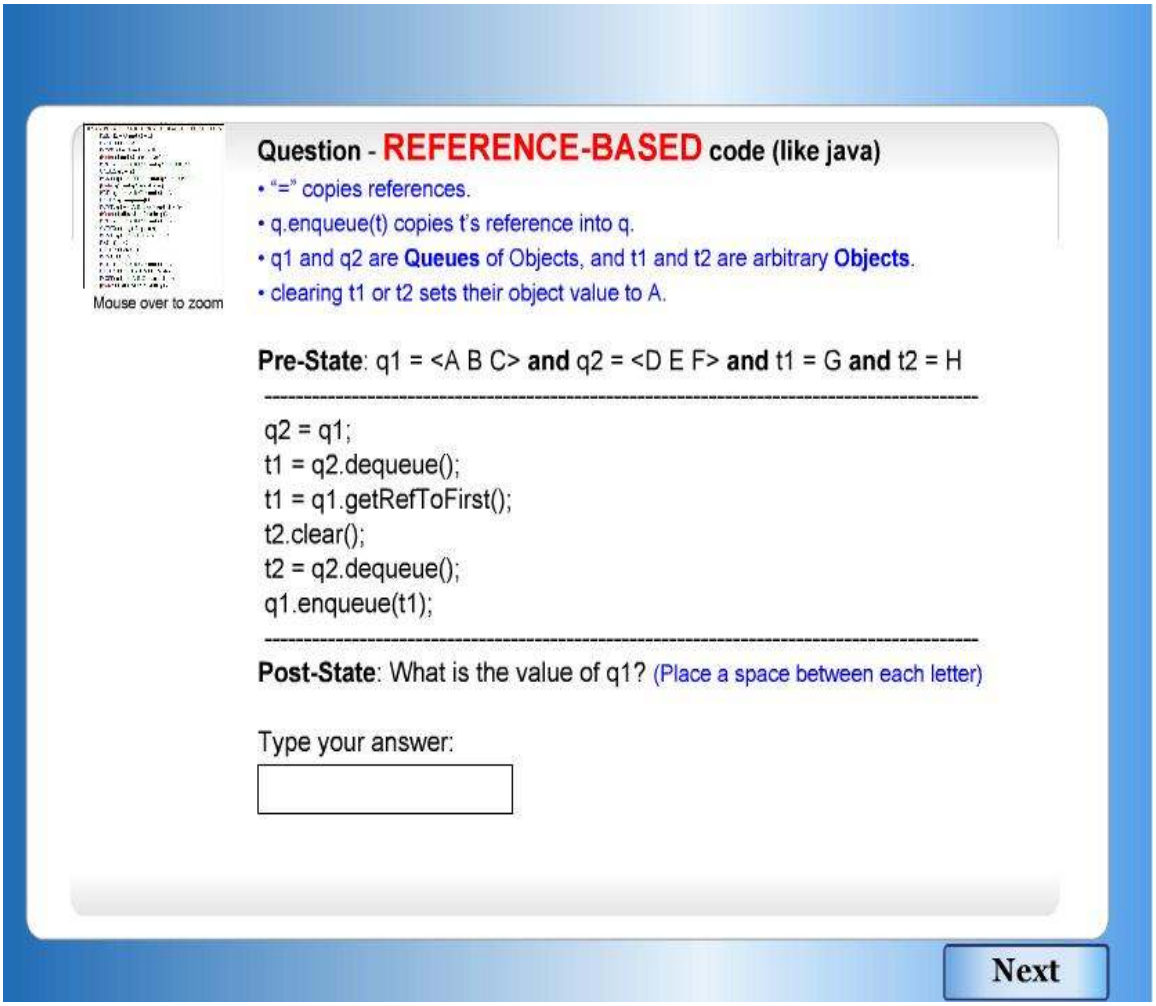

Figure 6.9 Final survey page with reference-based code

6.2 Results from final survey for combined data

The final surveys were distributed to 22 of the author's personal contacts. Each of them was given two surveys. 20 people took both surveys and two people took only one of the surveys for a total of 42 surveys taken.

Each survey contained three questions: one for each paradigm. Therefore, there were 42 questions for each paradigm. Table 6.1 summarizes the results of the surveys.

	Copy	**Swap**	**Reference**
Total number of questions	42	42	42
Number correct	24	17	7
Percent correct	57.14%	40.48%	16.67%
Avg. time (all) in seconds	177.24	177.92	130.96
Std. Dev. (all)	169.89	149.35	113.66
Avg. time (correct) in seconds	212.81	241.09	278.20
Std. Dev. (correct)	180.41	170.10	100.69

Table 6.1 Summary statistics for results of final surveys (combined data)

Copy-based questions resulted in the highest number of correct answers (24), followed by swapping-based questions (17). Reference-based questions resulted in the least number of correct answers, with only 7 correct answers in 42 questions. These results (in percentages) are illustrated graphically in Figure 6.10. The large differences in the percentages (copy-reference and swap-reference pairs) indicate that the results are consistent with the anecdotal evidence comparing the difficulty between working with value and reference semantics.

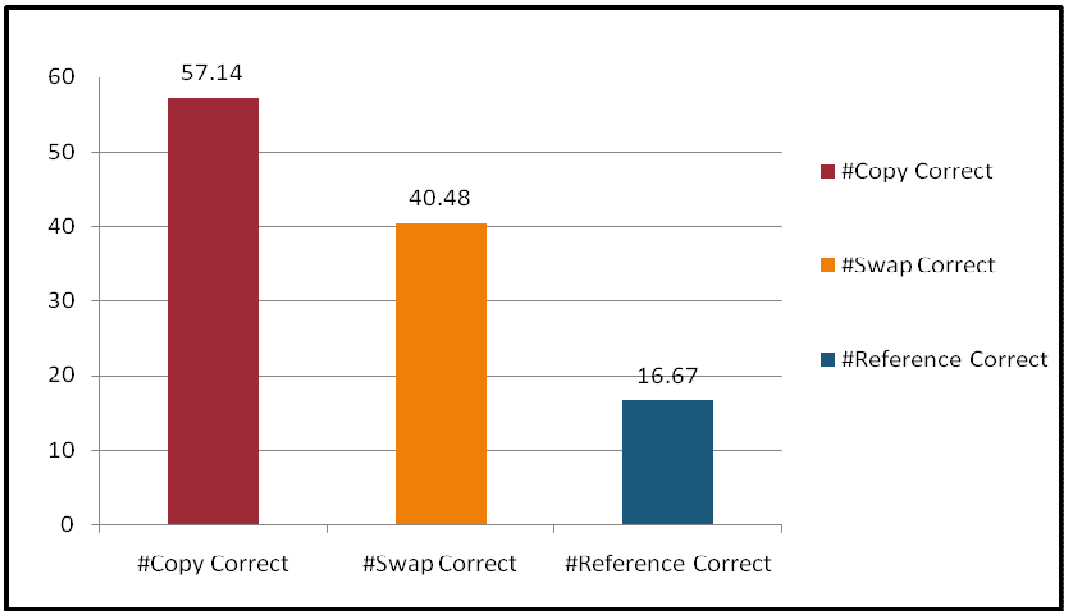

Figure 6.10 Bar-plot of percentage of number of questions answered correctly for combined data

The average time taken to answer copy-based questions (2 minutes, 57 seconds) and swap-based questions (2 minutes, 58 seconds) is higher than that taken to answer reference-based questions (1 minute, 11 seconds), a result we did *not* expect.

When we looked at the times taken to solve questions *correctly*, the results were very different than what we saw for the times taken to merely answer the question. Copy-based questions took the least amount of time on average (3 minutes, 33 seconds), then came swapping-based questions (4 minutes, 1 second), and finally reference-based questions (4 minutes, 38 seconds). Given that these results are so different from those that incorporated all times (both correct and incorrect answers) we can conclude that it took participants less time to answer questions incorrectly than it did for them to answer questions correctly – a result that is hardly surprising.

To check for normality of data, we plotted histograms for times taken to answer all correct questions in each paradigm category. Figures 6.11 through 6.13 show the histograms of times taken to correctly answer all copy, reference, and swap-based questions.

The data in the data in Figure 6.11 looks roughly normal, although the distribution is skewed to the left. The data in Figures 6.12 and 6.13 is uni-modal but not quite normal. We suspect that with larger numbers these data might begin to look normal also. However, since we do not have those numbers, we decided to analyze the data using robust non-parametric methods: we plotted the data using box plots with notches. The box plots give medians and 25th and 75th percentiles, and notches display the variability of the median between samples. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ. The width of a notch is computed so that box plots whose notches do not overlap have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. With data that appeared more normal we would use T-tests to determine significance.

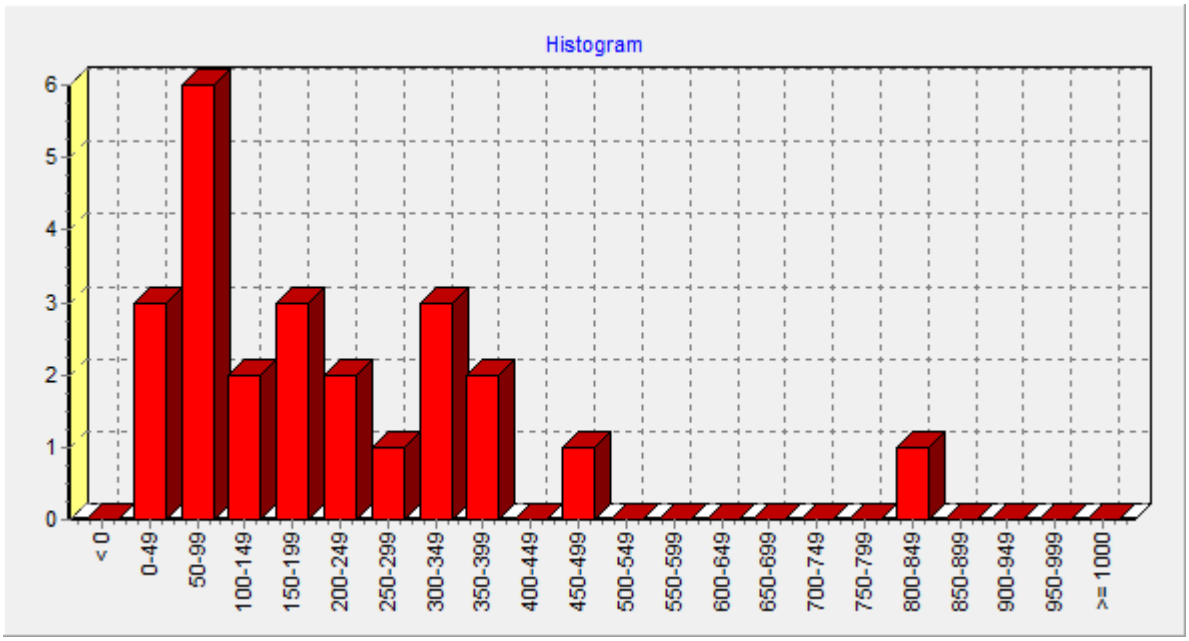

Figure 6.11 Histogram of answer times for correctly answered copying-based questions (combined data)

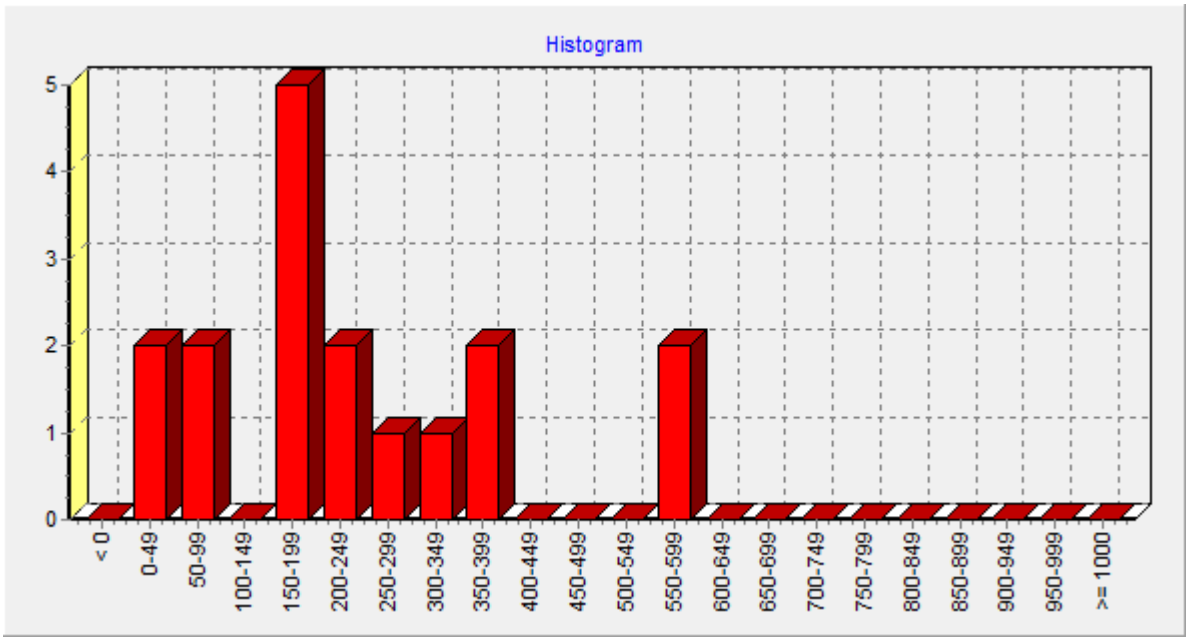

Figure 6.12 Histogram of answer times for correctly answered swapping-based questions (combined data)

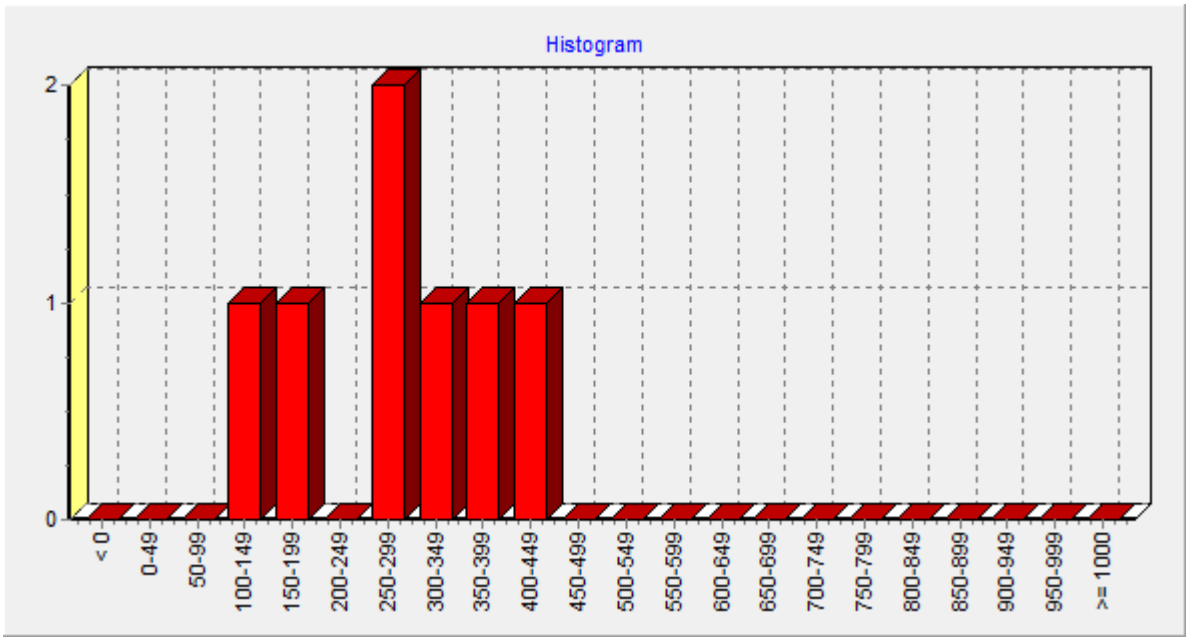

Figure 6.13 Histogram of answer times for correctly answered reference-based questions (combined data)

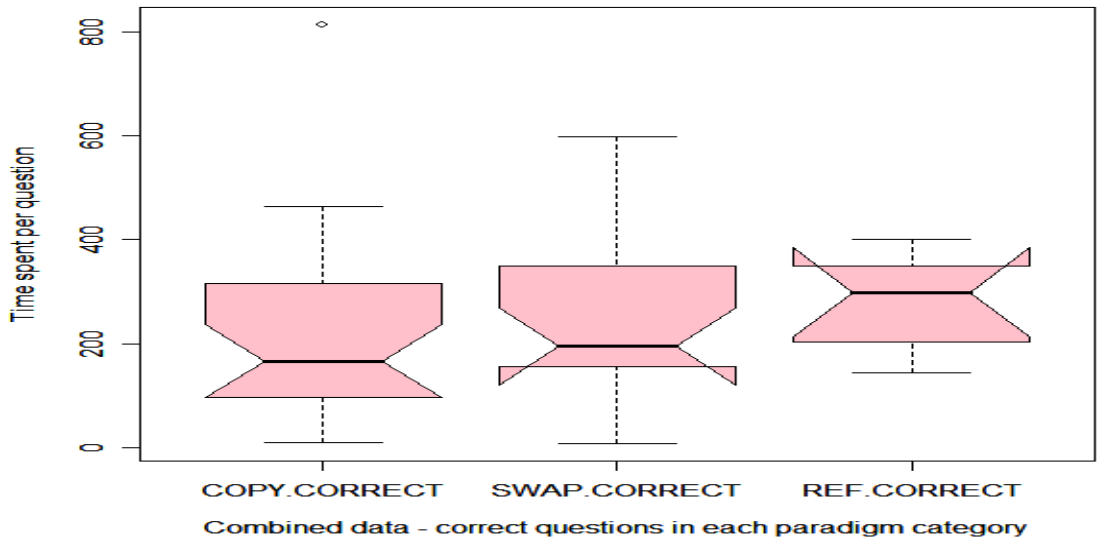

Figure 6.14 – Box plots with notches for times taken to correctly answer questions in each paradigm category (combined data)

The box plots with notches in figure 6.14 (details in Appendix table A.2) do not show any significant difference between the data for all paradigm pairs, since the notches for all plots overlap. Also, a single outlier is shown for the copy time plot.

6.2.1 A note on "non-attempts"

Initially we thought of trying to find a time threshold (for example, 5 seconds) for answering any question, below which it is considered a non-attempt. We would also include questions that were not answered or had a nonsensical answer as a non-attempt. In the end, we decided to not pursue this idea since the times always seemed arbitrary. Furthermore, the results obtained after doing the analysis did not impact the time to correctly answer questions. The reason for this is that it did not seem reasonable to have a threshold that would cause us to classify a correct answer as a non-attempt.

6.3 Comparison of results for differing Java experience

All the surveys were distributed to participants who varied in their Java programming experience. We divided these participants in three groups: Novices, Intermediates and Experts. We focused on Java experience because the code examples used Java-like syntax. In this section we compare look at how the results differ based on these three categories. From the total of 22 participants, 13 people considered themselves novices in Java, 5 considered themselves to have an intermediates level of experience, and 4 considered themselves to be experts. From the total of 42 questions answered in the surveys, 25 questions were answered by novices, 9 by intermediates, and 8 by experts.

	Copy			**Swap**			**Reference**		
Experience	**Novice**	**Inter.**	**Expert**	**Novice**	**Inter.**	**Expert**	**Novice**	**Inter.**	**Expert**
# Questions	25	9	8	25	9	8	25	9	8
# Correct	12	5	7	10	1	6	5	0	2
% Correct	48%	55.56%	87.50%	40%	11.11%	75%	20%	0%	25%
Avg. time (all)	138.09	229.09	241.68	174.36	88.15	290.04	118.49	68.04	240.32
Std. Dev. (all)	124.04	257.26	163.76	157.41	90.85	107.18	99.55	54.11	138.37
Avg. time (correct)	202.67	245.85	206.58	262.48	6.94	244.46	250.17	0.00	348.29
Std. Dev. (correct)	138.15	321.33	140.67	204.04	0.00	75.86	106.22	NA	44.08

Table 6.2 Summary statistics for results from final surveys for differing java experience

Table 6.2 gives the summary statistics of the results from the surveys distributed to participants belonging to different java experience categories. In the copy paradigm category, 12 questions were attempted correctly by novices, 5 by intermediates and 7 by experts.

In the swap paradigm category, 10 questions were attempted correctly by novices, 1 by intermediate and 6 by experts. Similarly, in the reference paradigm category, 5 questions were attempted correctly by novices, none by intermediates and 2 by experts. These numbers are better expressed as percentages, as in the bar charts in Figures 6.15 through 6.17. In general, the percentage of correct answers increased with experience, though those intermediate groups did worst in swapping and reference paradigms.

Interestingly, the average time spent on each question was longer for experts than for novices or intermediate programmers. Also, experts seemed to have spent more time on average to answer reference questions correctly than novices (intermediates did not answer any reference questions correctly).

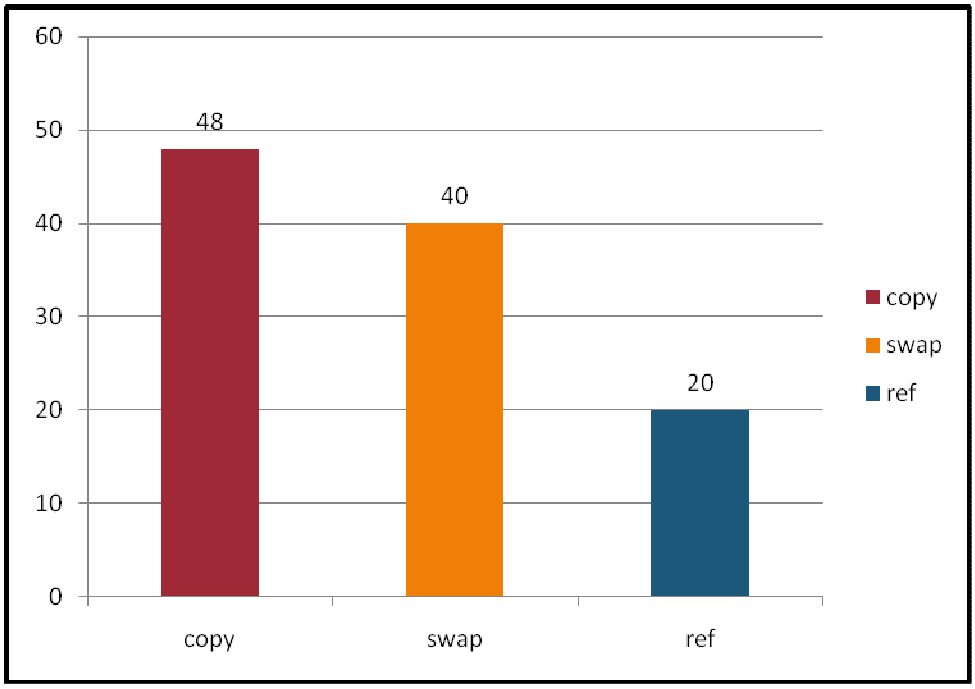

Figure 6.15 Bar graph of percentage of correct answers in novice group

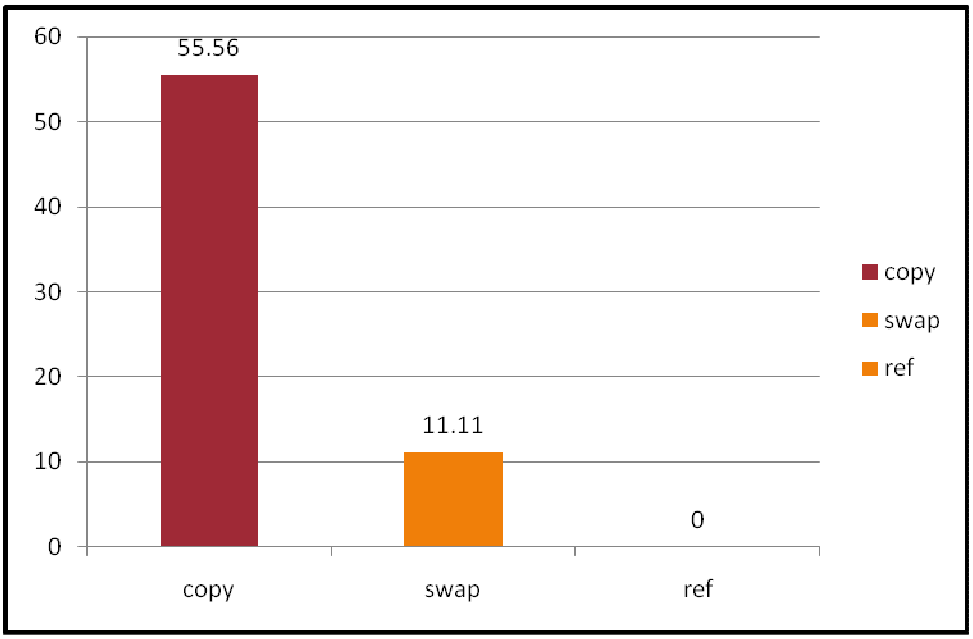

Figure 6.16 Bar graph of percentage of correct answers in intermediate group

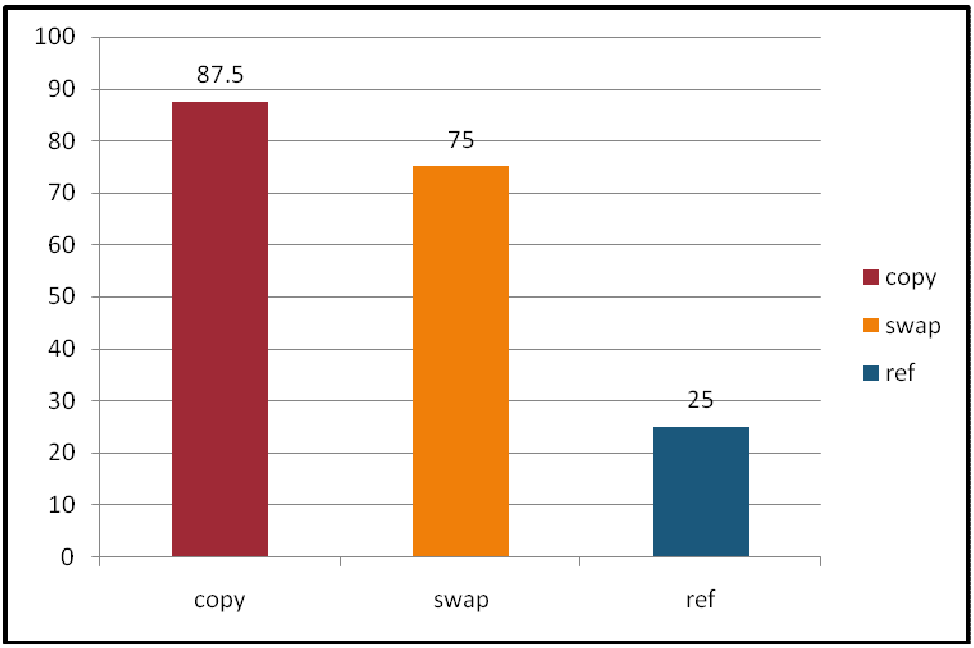

Figure 6.17 Bar graph of percentage of correct answers in expert group

The bar graphs in Figures 6.15 though 6.17 indicate the percentage of questions answered correctly for the different experience levels.

Within each experience level, a greater percentage of copy-based questions were answered correctly than swapping-based questions, and a greater percentage of swapping-based questions were answered correctly than reference-based questions. Within each paradigm, the expert programmers always answered the greatest percentage of questions correctly. Intermediate programmers answered a greater percentage of copy-based question correct than did novice programmers, but intermediate programmer did worse than novice programmers when it comes to the swapping paradigm or the reference paradigm. This result may be an aberration caused by the small number of intermediate programmers in the survey (four), or it may say something about the people who consider themselves intermediate level programs as compared to those who consider themselves novice programmers.

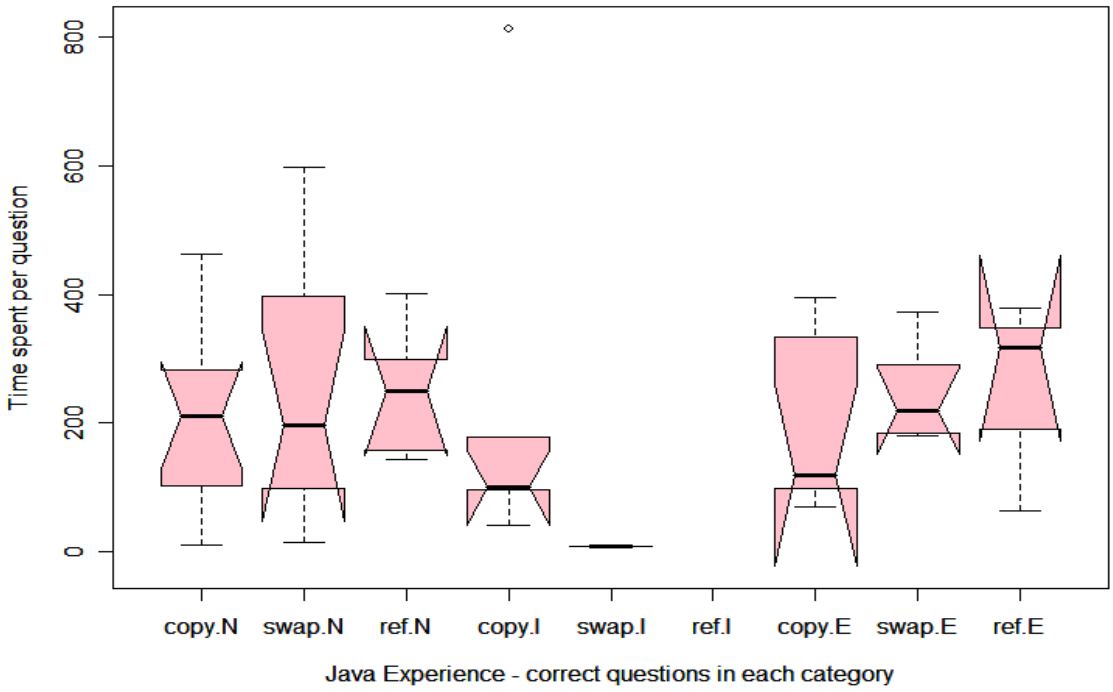

Figure 6.18 Box plots with notches for times spent per correctly answered question in each paradigm and Java experience category

Figure 6.18 shows the box plots with notches for times spent per correctly answered question in each paradigm and Java experience category. These box plot details can be found in Appendix A, Table A3.

6.4 Comparison of results of first and second surveys

Each participant of the final surveys was given two surveys, which we will call Survey 1 and Survey 2. Survey 1 was given before Survey 2. The questions in both surveys were randomly generated. The order of the questions was also randomly generated. In Survey 1, the first question was copy-based, the second was reference-based, and the third was swapping-based. In Survey 2, the paradigms appeared in the opposite order. Out of 22 participants, all 22 took Survey 1, and 20 took Survey 2. Table 6.7 gives summary statistics for each paradigm category for surveys 1 and 2.

	Copy		**Swap**		**Reference**	
Survey	**S1**	**S2**	**S1**	**S2**	**S1**	**S2**
Number of questions	22	20	22	20	22	20
Number correct	12	12	9	8	3	4
Percent correct	54.55%	60%	40.91%	40%	13.64%	20%
Avg. time	144.54	213.21	232.92	117.43	150.75	109.20
Std. Dev.	153.82	183.11	174.42	84.57	122.10	102.21
Avg. time (correct)	171.67	253.94	300.34	174.42	285.48	272.75
Std. Dev. (correct)	144.71	208.38	207.06	86.83	122.80	100.65

Table 6.3 Summary statistics of results from surveys 1 and 2

In general, Survey 2 has higher percentage of questions answered correctly in each paradigm category as compared to Survey 1. This is illustrated in Figure 6.19. The results are not surprising, since participants taking the second survey were able to become familiar with the contents and the interface of the first survey.

The average time taken to answer swap and reference questions is higher in Survey 1 as compared to that in Survey 2; but the average time taken to answer the copy questions is higher in Survey 2 as compared to Survey 1. This is true both for times to answer all questions and for times to correctly answer questions. Although, the participants have performed well in survey 2 as compared to survey 1, this may be the case because of an understanding bias that may have introduced amongst the participants. Since the participants have taken the surveys 1 and 2 in that order, they may have got a better understanding of the surveys questions and the paradigms. This may have lead to them performing better in the second survey. It may or may not have been the same case if the surveys were taken in reverse order.

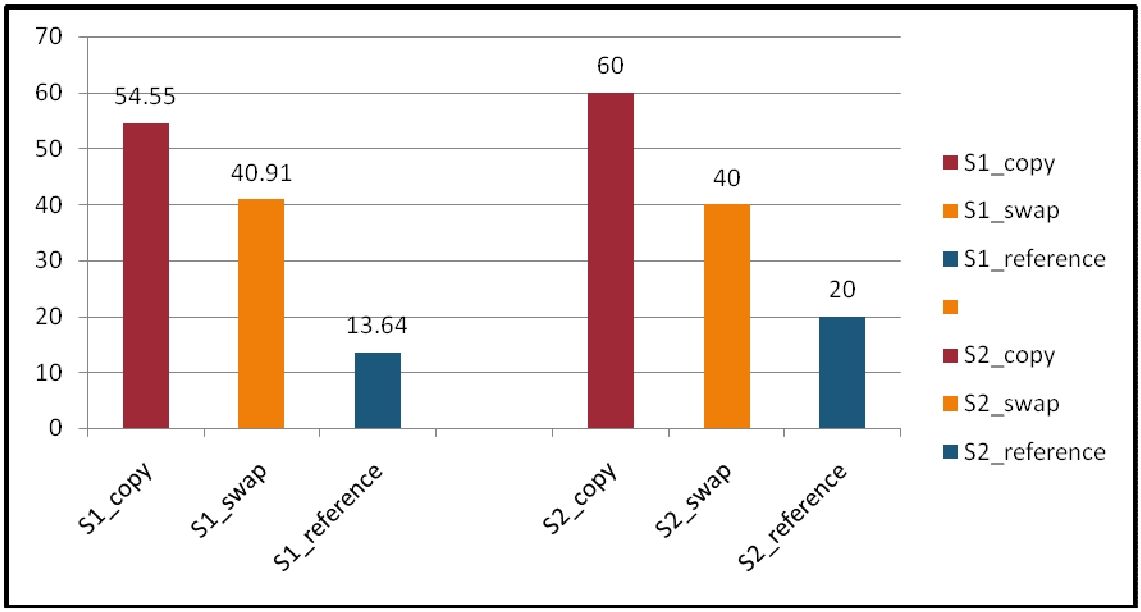

Figure 6.19 Bar graph of percent of number correct for different surveys (#1 vs. #2)

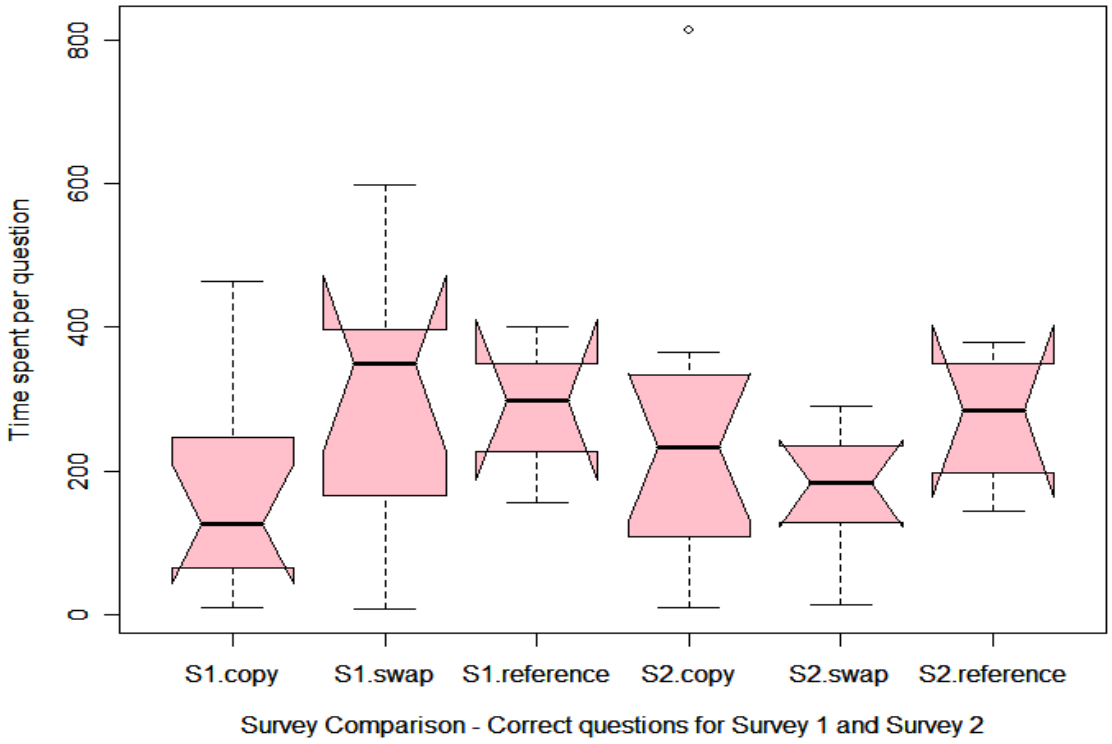

Figure 6.20 Box plots with notches of times for correct answers in surveys 1 and 2

Figure 6.20 gives the box plots with notches for times taken to correctly answer questions in each paradigm category for surveys 1 and 2. The details of this box plot can be found in Appendix A, Table A4. We can see a significant difference between the means of the paradigm pair copy-swap for survey 1, since the notches for both plots overlap.

6.5 Feedback from Final Surveys

This section presents the feedback from the final surveys. Feedback was elicited from six people who took the final surveys. Four of these were the same people who contributed feedback for the pilot survey.

Question #1 *How did you come up with answers to each question (using a scratch paper for calculations, computer, or just in the mind)?*

1. Just in mind
2. Scratch paper
3. First question of each survey on scratch paper and others in mind
4. Just in mind
5. Scratch paper
6. Using scratch paper and in mind both

Question #2 *Which question type was harder to solve and why?*

1. Ref was hard, been a long time since I worked on them.
2. Swap, since of the additional concept that T1 is cleared after every Swap enqueue.
3. Swap - Had to be very attentive
4. Reference
5. Swap
6. There was some confusion in the basic operations of swap, push and pop

Question #3 *Which question type was easiest to solve and why?*

1. Swap, no specific reason
2. Copy-Based; Because the answer was obvious especially after solving first question
3. Copy - no manipulation
4. Swap
5. Copy
6. Didn't find any question too easy to solve; all were of same difficulty level

Question #4 *Did you have any issues with your computer or other external hindrances/disturbances that hampered your continuity while taking the survey?*

1. No
2. No
3. Yes, kind of.
4. No
5. Yes, TV
6. No, I didn’t find any such issues

Question #5 *Did you have any confusion about the survey itself?*

1. No
2. No
3. Yes
4. No
5. A little
6. The purpose of the survey was not clear.

Question #6 *Did you find any ambiguity in the survey questions or concepts?*

1. No
2. No
3. Not really. Didn't get a clear idea of why to have different reference and copy
4. No
5. No
6. Yes there was confusion in the survey since the solved examples PDF stated complete opposite steps for swap - enqueue and dequeue in the queue. Also q1, q2 were tree objects, wasn’t clear till the end

Question #7 *Were you aware that the survey was timed and did the time factor influence your performance?*

1. I was not aware of that fact
2. Yes
3. I was aware but I did not get alert about time
4. No
5. Yes
6. The survey was timed and it would affect the performance wasn't mentioned anywhere.

Question #8 *What is your expertise level in java (novice, intermediate, expert)?*

1. Novice
2. Novice
3. Intermediate
4. Novice
5. Novice
6. Novice

Question #9 *When you solved the first question in the survey, did you easily anticipate the answers to other 2 questions without actually solving them?*

1. No
2. No
3. Yes
4. No
5. No
6. No

Question #10 *Did the code statement sequence or difficulty of the operations to be performed in each question affect your performance?*

1. No
2. No
3. Yes
4. No
5. No
6. No

Question #11 *What preparation was done by you before taking the survey (reading material, setup and so on)?*

1. No preparation
2. Read the documents provided on the website
3. No preparation
4. No preparation
5. No preparation
6. Read the examples and introduction on the webpage before giving the survey

Question #12 *How would you rate your performance in the surveys on the scale of 1 to 5 (5 being the highest)?*

1. 1
2. 4
3. 3
4. 4
5. 3
6. 2.5

Question #13 *Would you suggest any improvements in the presentation or contents of the survey?*

1. I had to browse to the link twice for taking the 2 surveys. It can be setup in such a way that it redirects back to the 2nd survey after the completion of 1st one.
2. No, everything was very well explained.
3. No, it seems fine after I got the result. Otherwise it did seem to be difficult.
4. It is pretty good.
5. Side by side examples instead of a constant need to zoom over the examples.
6. The confusion stated in 6 should be cleared. Whether the object is pushed in queue from left or right should be made a little clearer. Also the purpose of the survey can be explained a little.

The feedback from the final surveys was more positive than the feedback from the pilot surveys. Most of the participants found the copy-based and swapping-based questions easier than the reference-based questions, which was reflected in the actual data. They found the surveys self sufficient in terms of content and more informative. The presentation was more user-friendly and did not require the participants to refer to external documents for help.

7 Conclusions and Future Work

The research we conducted in this thesis gave us a better understanding of the impact of reference semantics on programmers. In general, we found results that met our broad expectations: questions involving copying-based code were the easiest to answer followed by questions involving swapping-based code, followed by questions involving reference-based code. The times spent by participants to achieve correct answers followed this same pattern. We were somewhat surprised that these results even held for experienced Java programmers, but as the number of people in that group was rather small, more investigation is needed. In general, a larger study and further investigation is needed to conclusively determine the impact of reference semantics on programming and reasoning. Neither of our hypotheses – more value-based questions would be answered correctly and in less time – were statistically validated.

Further work is also needed to eliminate potential biases that may have occurred in the research presented here. In the pilot surveys, some participants said that they found swapping questions the easiest, but the results did not reflect this – they indicated that those participants answered more copying-based questions correctly than swapping-based questions. This could indicate an expectation bias. Some participants may have been exposed to the swapping paradigm in the CS5704 course and felt that the professor expected them to do better on swapping-based questions. This bias might be mitigated if participants were not exposed to the swapping-based paradigm at all, or it future survey simply compared copy-based and reference-based paradigms.

Four out of the 22 participants of the final surveys were already part of the pilot surveys. Though their number is small, this may have introduced some understanding bias, since they were better acquainted with the questions and the paradigms from the pilot surveys. Since most participants had already been exposed to copying and reference paradigm through Java and other programming languages, this probably affected the swapping paradigm most.

The comparison we did between surveys one and two in the final survey did seem to indicate the understanding bias was present. In general, if participants' understanding increased for all three paradigms, the understanding bias should not have an effect on our hypotheses, but if participants understanding increased disproportionately for one paradigm over another, the hypotheses could be affected. Additional studies in which participants take multiple surveys could help determine the extent and the variance in this understanding bias.

We think the findings from this survey can reasonably be generalized to container data structures such as stacks, lists, sets, and maps. We would like to look at different collection data structures to verify this. In any case, the results cannot be generalized to entire programs. In many parts of a program, components that use value semantics and components that use reference semantics will behave the same because aliased variables are never modified. In other words, a portion of value-based code and a similar portion of reference-based code may yield the same state. We need empirical data to find the relative proportions of programs that exhibit different behavior due to reference semantics. This might entail looking at typical Java programs and rewriting them with value semantics to see where statements need to differ.

The surveys take dedication from the participant to complete. The fidelity hinges on how serious the users are about completing the survey. A classroom setting where scores correspond to grades can put some level of trust to the dedication of users. This may be a measure that can be taken in future, but it must be done in a class where these questions are relevant.

In general, none of our hypotheses were verified. We suspect that with larger sample sizes this may change. The data was suggestive of a trend and the questionnaires did indicate that survey participants found it easier to work with value semantics as compared to reference semantics.

While we would certainly like to administer the survey to a greater number of people, we feel that the results we obtained indicate that further research in this area is justified. One of our major contributions was developing a mechanism to automate the generation of survey questions. We would like to strengthen our method of distributing the surveys by taking advantage of the Random Question Generator we developed. One possible future direction would be to automatically generate random web-based survey questions so that distribution would be greatly simplified. Such a system might automatically collect and display data so that participants could see their results quickly. The lessens we learned and reported on here – from constructing the survey questions, from the mistakes in the pilot survey, and from the questionnaires – can help other researchers who intend to study these kinds of topics. While further investigation is clearly needed, we believe that this work represents an important first step in the empirical analysis of a topic that has previously only been discussed informally.

References

[CF91] Crank, E., and Felleisen, M., "*Parameter Passing and the Lambda Calculus*," Procs. 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1991, pp. 233-244.

[Cop91] Coplien, J. O., *Advanced C++ Programming Styles and Idioms*. Addison-Wesley. 1991.

[Cor01] *Common Object Request Broker Architecture* (CORBA), v2.4.2 February 2001.

[Gol96] Goldberg, B., *Functional Programming Languages*, ACM Computing Surveys, Vol. 28, No. 1, March 1996.

[HJ94] Hudak, P., Jones, M., *Haskell vs. Ada vs. C++ vs. Awk vs… An Experiment in Software Prototyping Productivity*, 1994.

[HLW92] Hogg, J., Lea, D., Wills, A., deChampeaux, D. and Holt, R. *The geneva convention on the treatment of object aliasing.* OOPS Messenger, *3* (2). pages 11-16. 1992.

[Hoa75] Hoare, C.A.R., *Recursive data structures.* International Journal of Computer and Information Sciences *4*, 2, pages 105-132, 1975.

[Hug90] Hughes, J., Why *Functional Programming Matters*, In D. Turner, editor, Research Topics in Functional Programming, Addison Wesley, 1990.

[HW91] Harms, D.E. and Weide, B.W. *Copying and swapping: Influences on the design of reusable software components.* IEEE Transactions on Software Engineering, 17 (5). pages 424-435. 1991.

[KC86] Khoshaflan, S.N. and Copeland, G.P., *Object identity*. In OOPSLA '86 Conference Proceedings, pages 406-416. ACM Press, 1986.

[Koe98] Koenig, A. and Moo, B., *Teaching standard C++*. JOOP 11, 7, 11–17. 1998.

[Kul04] Kulczycki G. W., *Direct Reasoning*, Doctoral Thesis, Clemson University, South Carolina, May 2004.

[KV06] Kulczycki, G and Vasudeo J., *Simplifying Reasoning about Objects with Tako*, 2006.

[LS99] Leino, K. R. M. and Stata, R., *Virginity: A contribution to the specification of objectoriented software.* Information Processing Letters 70, 2, 99–105. 1999.

[McL82] McLennan, B.J, *Values and Objects in Programming Languages*, SIGPLAN Notices, V17, #12, December 1982.

[MHW01] Murali, S., Long, E.T.M, Harner, J.,Weide, B.J, Wang, L., *A Formal Approach to Component-Based Software Engineering: Education and Evaluation*, IEEE, 2001.

[Nie89] Nierstrasz, O., *A Survey of Object-Oriented Concepts*, University of Geneva 1989.

[NL95] Ng, K.W. and Luk, C.K., *A Survey of Languages Integrating Functional, Object-oriented and Logic Programming*, 1995.

[NVP98] Noble, J., Vitek, J. and Potter, J. *Flexible alias protection.* Lecture Notes in Computer Science, 1445. pages 158-185. 1998.

[SAK00] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., *Reasoning about software-component behavior*. In Procs. Sixth Int. Conf. on Software Reuse. 2000.

[Vas06] Vasudeo, J.,*The Design and Implementation of the Tako Language and Compiler*, Master's Thesis, Virginia Tech, Virginia, May 2006.

[WH01] Weide, B.W. and Heym, W.D., *Specification and verification with references.* In Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems. 2001.

[WPH02] Weide, B.W., Pike, S.M. and Heym, W.D., *Why swapping?* In Proceedings RESOLVE 2002 Workshop, 2002.

Appendix A

We used box-and-whisker plots with notches to detect outliers and find significant difference between the sample pairs in the final survey results. Following is a description of the statistical data obtained by plotting box-plots of a given population of data:

$stats	A matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot. If all the inputs have the same class attribute, so will this component.
$n	A vector with the number of observations in each group.
$conf	A matrix where each column contains the lower and upper extremes of the notch.
$out	The values of any data points which lie beyond the extremes of the whiskers.
$group	A vector of the same length as out whose elements indicate to which group the outlier belongs.
$names	A vector of names for the groups.

Table A.1 Box-plots statistical data description

We used the statistical tool R to obtain box-plots of all data of the final surveys results. Following are the findings:

	Copy	**Swap**	**Reference**
Total number of correct questions	24	17	7
Lower whisker Lower hinge Mean Upper hinge Upper whisker	9.29 97.00 166.54 315.96 463.04	6.94 155.89 194.29 348.68 597.16	144.20 203.44 298.55 348.29 401.22
Lower extreme of notch Upper extreme of notch	95.92 237.15	120.41 268.17	212.04 385.05
Outlier	813.98		

Table A.2 Detail data for box-plots with notches in Figure 6.14 (final combined data – correct)

	Copy			Swap			Reference		
JAVA Experience Categories	**N**	**I**	**E**	**N**	**I**	**E**	**N**	**I**	**E**
Total # of Questions	25	9	8	25	9	8	25	9	8
# of Correct Answers	12	5	7	10	1	6	5	0	2
Lower whisker	9.29	40.98	69.84	14.44	6.94	180.35	144.20	NA	62.68
Lower hinge	102.24	96.80	97.34	99.10	6.94	184.70	156.66	NA	189.90
Mean	211.46	99.27	117.82	195.81	6.94	219.24	250.22	NA	317.13
Upper hinge	283.17	178.23	334.01	397.16	6.94	290.52	298.55	NA	348.29
Upper whisker	463.04	178.23	395.73	597.16	6.94	372.73	401.22	NA	379.46
Lower extreme of notch	128.94	41.73	-23.51	46.89	6.94	150.98	149.96	NA	172.64
Upper extreme of notch	293.98	156.81	259.16	344.73	6.94	287.50	350.48	NA	461.61
Outliers		813.98							

Table A.3 Detail data for box-plots with notches in Figure 6.18 (final – correctly answered questions for differing java experience categories)

	Copy		Swap		Reference	
Survey#	**S1**	**S2**	**S1**	**S2**	**S1**	**S2**
Total Number of Questions	22	20	22	20	22	20
Number of Correct Answers	12	12	9	8	3	4
Lower whisker	9.29	10.57	6.94	14.44	156.66	144.20
Lower hinge	64.56	108.55	165.42	127.50	227.61	197.21
Mean	126.02	233.03	348.68	182.53	298.55	283.67
Upper hinge	247.41	334.01	397.16	235.19	349.88	348.29
Upper whisker	463.04	364.19	597.16	290.52	401.22	379.46
Lower extreme of notch	42.62	130.20	226.63	122.37	187.01	164.31
Upper extreme of notch	209.42	335.87	470.72	242.69	410.09	403.03
Outliers		813.98				

Table A.4 Detail data for box-plots with notches in Figure 6.20 (final – correctly answered questions for comparison between surveys 1 and 2)

Appendix B

B.1 Source Code Design for Random Question Generator

The code to generate random code statements was developed in Java. The code is made up of 5 packages, each containing a set of related classes. These packages are:

- questiongenerators
- questionsolvers
- deepcopy
- outputs
- rng (random number generator)

The package *questiongenerators* contains the following classes:

SwapQuestionGenerator: contains the method "generateSwapQuestion" to generate a set of 5 code statements based on swap operation mode.

CopyQuestionGenerator: contains the method "generateCopyQuestion" to generate a set of 5 code statements based on copy operation mode.

RefQuestionGenerator: contains the method "generateRefQuestion" to generate a set of 5 code statements based on reference operation mode.

Other classes included in this package are Initialization, Record, QuestionTemplateGenerator, and Statement

The package *questionsolvers* contains classes the following classes:

SwapQSolver: contains the method "generateSwapSolution" to generate the post-state of both the queue and object variables based on swap operation mode, given their pre-state.

CopyQSolver: contains the method "generateCopySolution" to generate the post-state of both the queue and object variables based on copy operation mode, given their pre-state.

RefQSolver: contains the method "generateRefSolution" to generate the post-state of both the queue and object variables based on reference operation mode, given their pre-state.

Other classes included in this package are ListQueue, object, ObjectType, QuestionSolver and UnderflowException.

The package *deepcopy* contains classes such as *DeepCopy, FastByteInputArrayStream* and *FastByteOutputArrayStream* with methods to make a deep copy of any object using *java.io.InputStream* and *java.io.OutputStream* classes, which read from and write data to a buffer. Deep copying technique is used to implement the copying operations in value semantics.

The package *rng* (random number generator) contains the *Main* class from where the flow of the program progresses. It also contains *Constants* class that stores all the global constants / fixed variables whose values can be altered as required for effect throughout the program. All the randomization mentioned in the survey design is implemented by the Java class *java.util.Random*.

The software "Question Writer" (academic license) purchased from Central Question Ltd. was used to create the surveys. The main features that Question Writer surveys offered are:

- Facility to record time spent on each question in a survey
- Facility to record total time taken to complete a survey
- Facility to publish the surveys on the web
- Automated results forwarding to specified email address
- HTML based surveys that can be easily formatted
- Portable xml compilation