

Extreme Unit Testing: Ordering Test Cases to Maximize Early Testing

Allen Parrish

Dept. of Computer Science

The University of Alabama

Tuscaloosa, AL 35487 USA

+1 205 348 3749

parrish@cs.ua.edu

Joel Jones

Dept. of Computer Science

The University of Alabama

Tuscaloosa, AL 35487 USA

+1 205 348 1618

jones@cs.ua.edu

Brandon Dixon

Dept. of Computer Science

The University of Alabama

Tuscaloosa, AL 35487 USA

+1 205 348 0597

dixon@cs.ua.edu

ABSTRACT

eXtreme programming™ is one of several lightweight software development methodologies. It involves extremely short incremental release cycles, early and frequent testing, heavy use of design refactoring, and pair programming. Although early and frequent testing is very important, there is little guidance to date in terms of the specifics of the testing process. In this paper, we propose a new process model for what we call “ordered incremental testing,” where guidance on test order is provided. In both of these models, testing is conducted iteratively with very short iteration cycles as a single class is developed.

Keywords

Incremental testing, eXtreme programming, micro-development cycles, ordered incremental testing

1 INTRODUCTION

eXtreme programming™ (XP) is one of several lightweight software development methodologies. It involves extremely short incremental release cycles, early and frequent testing, heavy use of refactoring, and pair programming [1]. XP has had early success on its first project [3].

Testing is a particularly important part of XP. In particular, XP requires unit testing, with a strong emphasis on early and frequent testing during the development process. XP also requires that test cases be written prior to coding, and (typically) that test cases be written in a form where they can be automated. A popular testing tool is the JUnit suite for Java [2]. This tool eases the execution of unit tests. Its focus is to automate the process and to reduce the amount of manual output checking (i.e., GuruChecksOutput [5]).

Although XP contains an extensive testing philosophy, it does not specify the actual mechanics of testing in specific situations. Tools such as JUnit provide no guidance as to the order tests in which tests are to be implemented [2]. In

this paper, we consider the problem of developing and testing a single class. In XP, testing normally occurs as small subsets of methods for a class are developed.

We use the term *incremental testing* to refer to testing in multiple iterations during the development of a single class [7]. The goal of ordered-incremental testing is to reveal defects in the context of the simplest possible collection of methods by testing small numbers of methods first. We use the term *ordered incremental testing* to describe the overall XP testing methodology. In this paper, we present two separate process models for conducting testing. One such model involves developing test cases one at a time in an *ad hoc* order; the current state of the art. As each test case is developed, the methods needed to execute that test case are written, allowing the test case to be executed. The other model, *ordered incremental* involves developing larger sets of test cases and then using a simple greedy strategy to determine a desirable order in which to develop the methods in those test cases. This strategy attempts to maximize the amount of testing that can occur after each method is developed. This greedy strategy can be described as: “Write some tests, and write the method(s) that allow you to run more tests sooner.” We give an optional algorithm to formalize our ideas.

We present both of these process models to compare and contrast. Both have relative advantages and disadvantages, which we also discuss. To make our discussion concrete, our discussion takes place in the context of developing and testing an example ordered list class.

The remainder of the paper is organized as follows. In Section 2, we present our ordered list class along with some test cases that are appropriate for that class. We also define some basic testing terminology needed for the remainder of the paper. Sections 3 and 4 then present our two testing models. Section 5 contains our overall conclusion.

2 EXAMPLE AND BACKGROUND

Our discussion in this paper centers on an Ordered List class, where the elements of the list are integers and the list is maintained in ascending order. Figure 1 contains the method signatures for this class (written in Java).

```
public class OrderedList {
    public OrderedList() {}
    public boolean member(int v) {...}
    public OrderedList add(int v) {...}
    public int head() {...}
    public int length() {...}
    public boolean equals(OrderedList list) {...}
    public OrderedList tail() {...}
    public OrderedList delete(int v) {...}
}
```

Figure 1: Ordered List Class

This simple class contains several categories of relatively standard methods:

- A constructor (`OrderedList`) that returns an empty list;
- Methods to insert a new item at the appropriate location maintaining ascending order (`add`) and to delete an item by value (`delete`);
- Methods for returning the integer head of the list (`head`) as well as the list with the head removed (`tail`);
- Methods for returning the length of the list (`length`);

and to examine the list to determine the presence or absence of a particular item (`member`);

- A method to compare two lists for equality (`equals`).
- It is possible to identify a large number of test cases for this class. We design test cases as assertions on the relationship between objects produced by the various list methods. This is a relatively common XP practice, as it supports test automation [1,2]. Table 1 below contains a set of test cases for this class written as Java assertions, sometimes preceded with additional Java code. We do not claim that these test cases constitute adequate testing for this class; our interest here is to simply to define a large enough set to be interesting for a discussion of the incremental testing process. (Note that Table 1 uses the notation “<1,2>” to denote a list consisting of 1 as the first element and 2 as the second element. Also, for brevity, `Create` invokes the `OrderedList` constructor to generate a new, empty list object.)

In the remainder of the paper, we use this example (both the ordered list class and associated test cases) to explore our two process models for conducting incremental testing. Section 3 contains a relatively *ad hoc* model, while the model of Section 4 involves more extensive test planning prior to coding. Before discussing these models, however, we conclude this section with a brief review of the pair programming development cycle and an introduction to some testing terminology used in the remainder of the paper.

	Assertion	Explanation
1.	<code>assert (Create().add(1).head()==1);</code>	Adding 1 to \diamond and taking the head should return 1.
2.	<code>assert (Create().member(1)== false);</code>	1 is not a member of \diamond .
3.	<code>assert (Create().add(1).tail.equals(Create()));</code>	Adding 1 to \diamond produces $\langle 1 \rangle$. Taking the tail of $\langle 1 \rangle$ should produce \diamond .
4.	<code>OrderedList L = new OrderedList(); L = L.add(1); assert (L.add(2).head() == L.head());</code>	$L = \langle 1 \rangle$ (after the add on the second line). After adding a 2, returning the head of the list $\langle 1,2 \rangle$ should be equal to returning the head of L itself.
5.	<code>OrderedList L = new OrderedList(); L = L.add(2); assert (L.add(1).head() == 1);</code>	$L = \langle 2 \rangle$ (after the add on the second line). After adding a 1, returning the head of the list should be equal to 1 (the item just added).
6.	<code>OrderedList L = new OrderedList(); L = L.add(2); assert (L.add(1).tail().equals(L));</code>	$L = \langle 2 \rangle$ (after the add on the second line). After adding a 1, the list is $\langle 1,2 \rangle$; returning the tail is $\langle 2 \rangle$, which is equal to the original list L.
7.	<code>OrderedList L = new OrderedList(); L = L.add(0); assert (L.add(1).tail().equals(L.tail.add(1)));</code>	$L = \langle 0 \rangle$ after the add on the second line. After adding a 1, the list is $\langle 0,1 \rangle$; returning the tail yields $\langle 1 \rangle$. This is asserted to be equal to $L.tail(\diamond)$, followed by adding “1” (which yields $\langle 1 \rangle$).
8.	<code>assert (Create().add(1).member(1) == true);</code>	Adding 1 to \diamond and then doing a member test on 1 should return true.
9.	<code>assert (Create().add(1).member(2)== Create().member(2));</code>	Performing a member test for the existence of 2 in list $\langle 1 \rangle$ should be equal to performing a member test for existence of 2 in \diamond .
10.	<code>assert (Create().add(1).length() == Create().length + 1);</code>	Performing a length test on $\langle 1 \rangle$ should be 1 greater than a length test on \diamond .
11.	<code>assert (Create().delete(1).equals(Create()));</code>	Based on the spec (not provided here), deleting 1 from \diamond should be a no-op and simply produce \diamond .
12.	<code>assert (Create().add(1).delete(1).equals(Create()));</code>	Deleting 1 from $\langle 1 \rangle$ should equal \diamond .
13.	<code>assert (Create().add(1).delete(2).equals (Create.delete(2).add(1)));</code>	Deleting 2 from $\langle 1 \rangle$ should result in $\langle 1 \rangle$. Deleting 2 from \diamond and subsequently adding 1 should also result in $\langle 1 \rangle$.

Table 1: List Test Cases

We note that the overall pair programming development cycle may be characterized in five phases, with an iteration time through them measured in the ones to tens of minutes range [6]. The first phase is “micro-design,” where pairs determine the test cases and code to make the test cases pass. The second phase is to write one or more test cases. The third phase is to run the test cases, to make sure that they either fail to compile or fail to run. This phase may be iterated with debugging if the expected failure did not occur. The fourth phase is to implement the methods. The fifth phase is to run the test cases, iterated with debugging if the expected success did not occur. The micro-development cycle then proceeds starting with the second phase, writing more test cases, or by starting the process all over again with “micro-design.” We note that such a development cycle is not new with XP [4].

To clarify the subsequent discussion, we define some basic testing terminology that is consistent with the above development cycle. We say that a *test run* is the actual execution of a test case. Test runs are said to be *expected-positive* whenever it is expected that they will run properly (i.e., after the methods have been implemented). Test runs are said to be *expected-negative* whenever it is expected that they will not run properly (i.e., in the third phase above, before the methods have been implemented). A test run is said to be *successful* if its goal is met. That is, an expected-positive test run is successful if it *does not* reveal a defect; an expected-negative test run is successful if it *does* reveal a defect. A test run is *feasible* if it is possible to achieve success when executed, and *infeasible* if it is impossible to achieve success. For example, if one or more methods referenced by a test case have not been written yet, then an expected-positive test run of that test case is infeasible.

3 TESTING MODEL 1: AD HOC METHOD ORDERING

With this model, test cases are written one at a time, without regard to a specific overall ordering of test cases. Then once a particular test case is tested in an expected-negative test run, the methods referred to in the test case are implemented and the test case is once again executed (this time in an expected-positive test run). Once this test run is successful, the next test case is written, and an expected-negative test run is conducted with that test case. If any of the methods in this test case are not written yet, they are written now, and an expected-positive test run is conducted. In subsequent iterations after the first, it may not be necessary to implement all of the methods referred to in that iteration’s test case, as some of those methods may have been implemented in a previous iteration.

Assume that there are n test cases and k methods in the class under development. We note that the entire set of n test cases may not be known *a priori*, but may very well be developed individually as each is tested. Still, we assume

that the total number is finite (n). We can state this model as a relatively straightforward procedure:

For each test case $1 \dots n$:

- Write the test case (which references methods $m_i \dots m_j$).
- Conduct an expected-negative test run of this test case. Debug the class and repeat this test run until it is successful.
- For any method m in $m_i \dots m_j$ that has not previously been written, write method m .
- Conduct an expected-positive test run of this test case. Debug the class and repeat this test run until it is successful.

We now apply this testing model in the context of our `OrderedList` class, for which we identified 13 test cases in Table 1. As above, we do not assume that all of these test cases are known *a priori*. Since our process does not dictate a specific ordering of test case runs, we arbitrarily assume the same ordering as Table 1. We then have the following:

1. Develop and run test case 1 (`assert(Create().add(1).head()== 1);`)
 - Write test case 1, which references `OrderedList` (through `Create`), `add` and `head`.
 - Conduct an expected-negative test run of this test case. Debug the class and repeat this test run until it is successful.
 - Write these three methods (`OrderedList`, `add`, `head`).
 - Conduct an expected-positive test run of this test case. Debug the class and repeat this test run until it is successful.
2. Develop and run test case 2 (`assert(Create().member(1)== false);`)
 - Write test case 2, which references `OrderedList` (through `Create`) and `member`.
 - Conduct an expected-negative test run of this test case. Debug the class and repeat this test run until it is successful.
 - Write the `member` method. The `OrderedList` method was already written as part of the testing process for test case 1.
 - Conduct an expected-positive test run of this test case. Debug the class and repeat this test run until it is successful.
3. Develop and run test case 3 (`assert(Create().add(1).tail.equals(Create()));`)
 - Write test case 3, which references `OrderedList` (through `Create`), `add`, `tail` and `equals`.
 - Conduct an expected-negative test run of this test case. Debug the class and repeat this test run until it is successful.

- Write the `tail` and `equals` methods. The `OrderedList` and `add` methods were already written as part of the testing process for test cases 1 and 2.
- Conduct an expected-positive test run of this test case. Debug the class and repeat this test run until it is successful.

4. Continue this pattern for test cases 4 through 13.

Given the order of test cases presented in Table 1, we can determine the number of methods that must be implemented on a particular iteration prior to an expected-positive run for each test case. The number varies depending on how many methods have been implemented on a previous iteration. For example, if test case 1 is tested first, there are 3 methods referred to by the test case, and all 3 must be implemented prior to an expected-positive run of that test case. However, if test case 2 is tested next (as above), then there are 2 methods referred to by the test case, but only 1 must be implemented in this iteration (as the constructor was already developed in the previous iteration). Table 2 lists the methods referenced by each test case, and the methods that must be developed as each test case is run, assuming this particular ordering of test runs.

Ideally, it would be nice to minimize the number of methods that are developed at each iteration. The smaller the number of methods that are introduced new as each test case is run, the easier it will be to pinpoint errors. In this particular example, only two test cases could be completed after the first four methods were developed. Two additional methods had to be written before test case 3 could be completed, at which point test cases 4 through 9 could also be completed. Of course, the number of methods that must be developed at each iteration is heavily dependent on the order in which the test cases are run. However, there is nothing in this particular process model that dictates this order.

A second issue is that test cases should ideally be run as early in the development process as possible. In this example, test cases 8 and 9 could be run immediately after test case 2, immediately after `member` was written. By running these test cases earlier, defects in the methods they refer to (such as `add`, which is heavily referred to by almost all of the remaining test cases) could be exposed in the context of a smaller number of methods. Again, this simpler context could make it easier to pinpoint errors. In the next section, we consider an alternative testing model that specifies the ordering of test case runs and (correspondingly) method development.

	Test Case	Methods Referenced	Methods Developed
1.	<code>assert(Create().add(1).head() == 1);</code>	<code>OrderedList, add, head</code>	<code>OrderedList, add, head</code>
2.	<code>assert(Create().member(1) == false);</code>	<code>OrderedList, member</code>	<code>member</code>
3.	<code>assert(Create().add(1).tail.equals(Create()));</code>	<code>OrderedList, add, tail, equals</code>	<code>tail, equals</code>
4.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(1);</code> <code>assert(L.add(2).head() == L.head());</code>	<code>OrderedList, add, head</code>	
5.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(2);</code> <code>assert(L.add(1).head() == 1);</code>	<code>OrderedList, add, head</code>	
6.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(2);</code> <code>assert(L.add(1).tail().equals(L));</code>	<code>OrderedList, add, tail, equals</code>	
7.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(0);</code> <code>assert(L.add(1).tail().equals(L.tail.add(1));</code>	<code>OrderedList, add, tail, equals</code>	
8.	<code>assert(Create().add(1).member(1) == true);</code>	<code>OrderedList, add, member</code>	
9.	<code>assert(Create().add(1).member(2) == Create().member(2));</code>	<code>OrderedList, add, member</code>	
10.	<code>assert(Create().add(1).length() == Create().length + 1);</code>	<code>OrderedList, add, length</code>	<code>length</code>
11.	<code>assert(Create().delete(1).equals(Create()));</code>	<code>OrderedList, delete, equals</code>	<code>delete</code>
12.	<code>assert(Create().add(1).delete(1).equals(Create()));</code>	<code>OrderedList, add, delete, equals</code>	
13.	<code>assert(Create().add(1).delete(2).equals(Create().delete(2).add(1));</code>	<code>OrderedList, add, delete, equals</code>	

Table 2 – Methods Developed Before Each Test Run (arbitrary *ad hoc* Ordering)

4 TESTING MODEL 2: OPTIMIZED METHOD ORDERING

Here we consider a second testing model that specifies the ordering of test case runs and method development. The basic idea is that there is a set (of size greater than 1) of test cases that are initially identified. Frequently, this might be the entire set of all test cases for the class, although not necessarily. Once a set of test cases is identified, then an attempt is made to order the test case runs in such a way that early testing is maximized. This means that defects are potentially revealed in the context of as few methods as possible, making those defects easier to localize.

To understand this approach, we first consider a procedure that produces an ordering in which methods should be developed (called a *development ordering*), based on the test cases in which those methods appear. Our procedure attempts to maximize the number of opportunities to test as methods are developed. These opportunities are called *test points*. More specifically, a development ordering is just an ordered sequence of methods; a test point is a position following a method m within a development ordering, where an expected-positive test run of a particular test case is feasible following m , but not following any method in the ordering prior to m . At each test point, we also attempt to maximize the number of test runs that can be conducted. In this way, we attempt to maximize the amount of meaningful testing that can be done early in the development process.

One algorithm to produce optimal development orderings would be to generate all possible orderings (all permutations of methods) and then count the number of test points for each ordering as well as the number of test cases for which expected-positive test runs are feasible at each test point. Since this algorithm is NP-complete, we propose a greedy heuristic. Specifically, to choose the next method in a development ordering, we first identify all methods that if chosen next, at least one expected-positive test run is feasible. Of these methods, we select the method with the highest *impact score*. The impact score attempts to quantify which method provides the most "progress" toward making test runs feasible for additional test cases. If there is no method that will result in a test point, we choose the method with the highest impact score among all methods that have not been previously chosen.

We compute the impact score based on the number of test cases by which the method is referenced, weighted for each test case. The weight for a given test case is based on the inverse of the number of methods that appear in that test case. For example, suppose a method M appears in two test cases, A and B . Moreover there are three methods referenced by test case A and two methods referenced by test case B . The impact score for method M is $1/3 + 1/2$ ($0.33 + 0.50$), which equals 0.83 . (Note that multiple references to a method within the single test case are

ignored in computing weights.)

Our `OrderedList` example will clarify the use of this heuristic. Our objective in this example is to define a development ordering among all methods in the `OrderedList` class of Figure 1, based on the test cases that appear in Table 1. Our first step is to determine, for each method, whether or not the selection of that method will result in a test point. There is no method that we can choose initially that will make an expected-positive test run feasible (for any test case). Thus, we must look at the impact scores for all methods. Table 3 contains the impact scores for all methods in the `OrderedList` class. Recall that `Create` is a method that invokes the `OrderedList` constructor, and so we compute the impact score for `OrderedList` based upon the appearance of `Create` as well as direct references to `OrderedList` in the various test cases.

Method	Impact Score
<code>OrderedList</code>	4.08
<code>add</code>	3.25
<code>head</code>	1.00
<code>member</code>	1.16
<code>tail</code>	0.75
<code>equals</code>	1.58
<code>length</code>	0.33
<code>delete</code>	0.83

Table 3 – Initial Impact Scores

The constructor method (`OrderedList`) has the highest impact score, so it is chosen as the first method in the development ordering. This is also the intuitive choice, as it is clear that no expected-positive test runs are feasible until this method is written.

To find the next method, we first determine if the selection of any particular method results in a test point. Because of test case 2 (`assert(Create().member(1) == false)`), selecting `member` next will result in a test point (allowing a feasible expected-positive run for test case 2). The `member` method is the only method that will result in a test point and is thus chosen next. Similarly, because of test cases 8 and 9 (e.g., `assert(Create().add(1).member(1) == true)`), selecting `add` next will result in a test point, and is again the only such method. Thus, our development ordering so far is: `OrderedList` \rightarrow `member` \rightarrow `add`.

Now we have two possible methods that will result in test points if chosen next: `head` and `length`. We therefore choose the method with the higher impact score. Since our previously computed impact scores from Table 3 were influenced by methods that have already been chosen (and are therefore now irrelevant), we thus re-compute the impact scores with `OrderedList`, `member` and `add` removed from consideration. The `head` method appears in 3 test cases, and is by itself in those test cases (other than

the already chosen methods). Thus, its impact score is 3. On the other hand, `length` only appears in one test case (also by itself except for already chosen methods); thus, its impact score is 1. As such, `head` is chosen next.

Once `head` is chosen, no additional methods besides `length` will result in a test point. So `length` is chosen next. Thus, our development ordering so far is: `OrderedList` → `member` → `add` → `head` → `length`.

At this point, `tail`, `equals` and `delete` remain to be chosen. None of these methods, if chosen next, will result in a test point. Once again, we must re-compute the impact scores for these three methods with methods already in the development ordering removed from consideration. The recomputed impact scores are given in Table 4.

Method	Impact Score
<code>tail</code>	1.5
<code>equals</code>	3.0
<code>delete</code>	1.5

Table 4 – Recomputed Impact Scores

The `equals` method has the highest impact score and is chosen next. Choosing `equals` allows either `tail` or `delete` to result in a test point if chosen next. As both appear in the same number of test cases (3), the choice for

the next method is arbitrary. We choose `delete` next, followed by `tail`. Thus, our final development ordering is: `OrderedList` → `member` → `add` → `head` → `length` → `equals` → `delete` → `tail`.

Once the optimal development ordering for the methods is known, then it is straightforward to get an analogous optimal ordering for test cases. In particular, it is straightforward to determine which test cases have feasible expected-positive test runs after each method in the development ordering. Table 5 for this model is the analog of Table 2 for the *ad hoc* model; it contains the 13 test cases from Table 1 in an order consistent with the development ordering for the methods. Each test case is accompanied by the methods referenced in that test case along with those methods that must be developed to make an expected-positive test run of that test case feasible. If no methods are listed beside a particular test case, then an expected-positive run of the test case is immediately feasible after the previous test case is run. Our heuristic attempts to minimize the number of methods that must be developed for each test case. As Table 5 shows, no more than two methods at a time have to be written prior to conducting some testing. For most test cases, an expected-positive test run is feasible after writing at most one method.

	Test Case	Methods Referenced	Methods Developed
2.	<code>assert(Create().member(1) == false);</code>	<code>OrderedList, member</code>	<code>OrderedList, member</code>
8.	<code>assert(Create().add(1).member(1) == true);</code>	<code>OrderedList, add, member</code>	<code>add</code>
9.	<code>assert(Create().add(1).member(2) == Create().member(2));</code>	<code>OrderedList, add, member</code>	
1.	<code>assert(Create().add(1).head() == 1);</code>	<code>OrderedList, add, head</code>	<code>head</code>
4.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(1);</code> <code>assert(L.add(2).head() == L.head());</code>	<code>OrderedList, add, head</code>	
5.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(2);</code> <code>assert(L.add(1).head() == 1);</code>	<code>OrderedList, add, head</code>	
10.	<code>assert(Create().add(1).length() == Create().length + 1);</code>	<code>OrderedList, add, length</code>	<code>length</code>
11.	<code>assert(Create().delete(1).equals(Create()));</code>	<code>OrderedList, delete, equals</code>	<code>equals, delete</code>
12.	<code>assert(Create().add(1).delete(1).equals(Create()));</code>	<code>OrderedList, add, delete, equals</code>	
13.	<code>assert(Create().add(1).delete(2).equals(Create().delete(2).add(1)));</code>	<code>OrderedList, add, delete, equals</code>	
3.	<code>assert(Create().add(1).tail().equals(Create()));</code>	<code>OrderedList, add, tail, equals</code>	<code>tail</code>
6.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(2);</code> <code>assert(L.add(1).tail().equals(L));</code>	<code>OrderedList, add, tail, equals</code>	
7.	<code>OrderedList L = new OrderedList();</code> <code>L = L.add(0);</code> <code>assert(L.add(1).tail().equals(L.tail.add(1)));</code>	<code>OrderedList, add, tail, equals</code>	

Table 5 - Methods Developed Before Each Test Run (Optimized Ordering)

Our example assumes that the development orderings for both methods and test cases are computed once for the entire class. In fact, such orderings may be computed on subsets of methods and test cases. For example, given the “on demand” nature of XP, it may be undesirable to focus on an entire class in this fashion. Instead, it may be desirable to identify and develop test cases in groups of 10 (say), and apply the development ordering heuristic to corresponding subsets of the methods.

Also, we have developed a tool that implements the above heuristic [7]. Thus, application of this approach does not require large amounts of manual effort. With small numbers of methods and test cases, the proper selection of the next method to be developed or the next test case to be executed may be obvious without a tool. Our tool has been applied to a number of different data structures-based classes, including a text editor class, as well as classes implementing various kinds of lists, stacks, queues, complex numbers and arrays.

There are pros and cons when comparing this model with the *ad hoc* model of Section 3. In particular, the *ad hoc* model requires no *a priori* test planning beyond the current test cases. Test cases can be identified and tested one at a time, on an on-demand basis. This allows the class design to change very flexibly as code is refactored, without having to discard test cases that are no longer usable.

On the other hand, as noted at the end of Section 3, the *ad hoc* model does not attempt to maximize the amount of testing performed early in the development process. In particular, depending on the order in which test cases are developed and tested, it may be necessary to develop several different methods before any testing can be performed. For example, if a test case references four methods, it may be necessary to develop all four methods in a given iteration before any testing of those methods is performed. In particular, if test case 13 from Table 1 were developed first, then it would be necessary to write the `OrderedList` constructor, `add`, `delete` and `equals` before *any* testing (other than expected-negative tests) could be performed. Similarly, without some attention to the overall ordering, test cases might be unnecessarily deferred. For the `OrderedList` example (and as noted at the end of Section 3), test cases 8 and 9 could be run immediately after test case 2.

5 CONCLUSION

In this paper, we have identified two process models for conducting incremental testing (i.e., testing with short

iteration cycles). The first model does not dictate an ordering for constructing methods or for developing and running test cases, but is specific regarding the structure of test cases and the steps required at each iteration. This model could be viewed as a formalization of the process described in [2].

The second model requires the identification of multiple test cases at each iteration of the development cycle, and then provides guidance as to the order in which those test cases are tested and their methods implemented. With this model, 7 test cases could be run as the first 5 methods are developed; in contrast, only 2 test cases could be run as the first 5 methods are developed with the ordering that we chose under the *ad hoc* model. Our heuristic simply and efficiently gives an ordering over the set of possible methods that is consistent with the XP practice “Do the simplest thing that could possibly work.” As there is a sacrifice in flexibility, however, there may be situations where the *ad hoc* model is more appropriate. With this paper, we hope to stimulate dialog regarding the relative suitability of the two models in different situations.

REFERENCES

1. Beck, K. *eXtreme Programming Explained*, Addison-Wesley, 2000.
2. Beck, K. and E. Gamma, “Test Infected: Programmers Love Writing Tests,” *Java Report*, vol. 3, no. 7, 1998, pp. 37-50.
3. C3 Team, “Chrysler Goes to ‘Extreme’”, *Distributed Computing*, October 1998, pp. 24-28.
4. Coad, P. and J. Nicola, *Object-Oriented Programming*, Yourdon Press, New York, 1993.
5. Farrell, J., “Tactical Testing,” <http://c2.com/cgi/wiki?TacticalTesting>.
6. Martin, R. and R. Koss, “Engineering Notebook: An eXtreme Programming Episode,” *C/C++ Users Journal* (Online only), <http://www.cuj.com/experts/1902/martin.htm>, vol. 19, no. 2, February 2001.
7. Parrish, A., D. Cordes and D. Brown, “An Environment to Support Micro-Incremental Class Development,” *Annals of Software Engineering*, vol. 2, 1996, pp. 213-236.