Compared methods

COMPUTER SCIENCE 511 (SPRING TERM, 2007)

OBJECT-ORIENTED SOFTWARE PRINCIPLES

Solutions for Mid-Term Examination

(15 points) **Problem 1. Comparing method specifications.** Six multiple-choice questions. You may explain each answer in one line, if you wish.

1. Which of the following is the easiest precondition to satisfy?
   
   (a) **REQUIRES**: arguments are non-null.
   
   (b) **REQUIRES**: false.
   
   (c) **REQUIRES**: true.
   
   (d) None of the above.

   **Answer**: (c), which makes no demands on the caller. To make a method work properly, a client (the caller) has to satisfy the method’s precondition, i.e., to make it true. The implementation (the callee) gets to assume that the precondition is already satisfied by the caller.

2. Which of the following method specifications is stronger?

   ($) **REQUIRES**: $x >= 0$
   
   ($$) **REQUIRES**: $x > 0$

   (a) ($) is stronger than ($$),
   
   (b) ($$) is stronger than ($),
   
   (c) ($) and ($$) are equally strong,
   
   (d) ($) and ($$) are incomparable.

   **Answer**: (a) ($) is stronger than ($$) because ($) requires less than ($$).

3. Which of the following method specifications is stronger?

   ($) **MODIFIES**: nothing
   
   ($$) **MODIFIES**: $x$

   (a) ($) is stronger than ($$),
   
   (b) ($$) is stronger than ($),
   
   (c) ($) and ($$$) are equally strong,
   
   (d) ($) and ($$$) are incomparable.

   **Answer**: (a) ($) is stronger than ($$$) because ($) modifies less than ($$$).

4. Which of the following method specifications is stronger?

   ($) **REQUIRES**: $x < 0$
   
   EFFECTS: returns a value > 0
   
   ($$) **REQUIRES**: $x <= 0$
   
   EFFECTS: returns a value >= 0

   (a) ($) is stronger than ($$$),
   
   (b) ($$$) is stronger than ($),
   
   (c) ($) and ($$$) are equally strong,
   
   (d) ($) and ($$$) are incomparable.

   **Answer**: (d) ($) and ($$$) are incomparable.
5. Suppose that the specification of \texttt{Ram.foo()} is stronger than the specification of \texttt{Bar.foo()}. A client program that works when it calls \texttt{Bar.foo()}:

(a) will work if it calls \texttt{Ram.foo()},
(b) will not work if it calls \texttt{Ram.foo()},
(c) may or may not work if it calls \texttt{Ram.foo()},
(d) none of the above.

\textbf{Answer:} (a) will work if it calls \texttt{Ram.foo()}.

6. Suppose that the specification of \texttt{Ram.foo()} is stronger than the specification of \texttt{Bar.foo()}. A client program that works when it calls \texttt{Ram.foo()}:

(a) will work if it calls \texttt{Bar.foo()},
(b) will not work if it calls \texttt{Bar.foo()},
(c) may or may not work if it calls \texttt{Bar.foo()},
(d) none of the above.

\textbf{Answer:} (c) may or may not work if it calls \texttt{Bar.foo()}.

(10 points) \textbf{Problem 2. Defective method specification.} Consider the specification of the method \texttt{"put"} from the Java library class \texttt{HashMap}:

\begin{verbatim}
    public Object put (Object key, Object value)
    // REQUIRES: "value" is not "null"
    // EFFECTS: inserts the pair ("key", "value") into the mapping, overriding
    // any existing mapping for "key", and returns previous value associated
    // with "key", or "null" if there was no mapping for "key".
\end{verbatim}

(This specification is a paraphrase of what is actually written for the method \texttt{"put"} in the documentation for Java 1.4.)

1. Explain what is bad with this specification, in no more than 3-4 lines.

\textbf{Answer:} The precondition here (the \texttt{REQUIRES} clause) does not rule out \texttt{"null"} as a value, so the hash map can store \texttt{"null"} values. But the postcondition (the \texttt{EFFECTS} clause) uses \texttt{"null"} as a special return value for a missing key. This means that if \texttt{"null"} is returned, you cannot tell whether the key was not bound previously or whether it was in fact bound to \texttt{"null"}. This is a bad design. And what is bad about it is that the return value is useless unless you know that no \texttt{"null"} values were inserted.

2. Propose an adjustment that will improve the specification.

\textbf{Answer:}

\begin{verbatim}
    public Object put (Object key, Object value)
    // REQUIRES: "value" is not "null"
    // EFFECTS: inserts the pair ("key", "value") into the mapping,
    // overriding any existing mapping for "key", and
    // returns previous value associated with "key", or
    // "null" if there was no mapping for "key".
\end{verbatim}
(5 points) **Problem 3.** *Deterministic method versus underdetermined method specification.* Can a deterministic method satisfy an underdetermined specification? If so, give an example. If not, explain in at most 5 lines.

**Answer:** The answer is YES. Sidebar 3.2 of [PDJ] states, “A procedure is underdetermined if for certain inputs its specification allows more than one possible result” and “An implementation of a procedure is deterministic if, for the same inputs, it always produces the same result.” Even though the specification allows more than one possible result, the procedure only needs to produce one of them and can therefore be written as a deterministic procedure. Here is an example:

```
// RETURNS: an arbitrary int
int foo ( ) { return 3; }
```

(5 points) **Problem 4.** *Abuse of exceptions.* Briefly (in at most 2-3 lines) give two situations in which a **REQUIRES** clauses is preferable to specifying that an exception is thrown.

**Answer:** (1) If a property is very expensive to check, then it is better (except during debugging) to require this property to be true (by including it as a precondition) than to check it in order to ensure that the right exception is thrown. (2) If a method is private and/or meant to be called from strictly controlled situations in which the property has already been checked and/or established, then it is not advantageous to clutter the code with exceptions that will never be thrown by correct executions.

(5 points) **Problem 5.** *Abuse of exceptions.* Consider the code fragment below. Explain what is wrong with it, in at most 2-3 lines. Ignore all issues related to efficiency and assume the code compiles without problem, where **Foo** is a user-defined class which includes a method called **g**().

```
try {
    Iterator t = collection.iterator ();
    while (true) { Foo foo = (Foo) t.next (); foo.g ( ); }
}
```

**Answer:** There is no way of knowing if **NoSuchElementException** is thrown by the iterator (having exhausted all elements in the collection) or by the method **g**(). The answer here follows the explanation given in Handout 12, pages 17-18.
(20 points) **Problem 6.** *Representation invariants.* Below is a (possibly defective) fragment of an implementation of a class “Set”, with a method “choose” that removes an arbitrary element from the set and returns it, and a method “add” that adds an element to the set:

```java
class Set {
    private Object[] elements;
    private int size;
    // constructor
    Set(int capacity) {
        elements = new Object[capacity];
        size = 0;
    }
    // methods
    Object choose() {
        if (size == 0) return null;
        size = size - 1;
        return elements[size];
    }
    void add(Object x) {
        this.element[size] = x;
        size = size + 1;
    }
    ...
}
```

Consider the 5 following candidates for a representation invariant. For each, state whether it is preserved by the method “choose”, the method “add”, by both methods or by neither, together with a justification of at most 2 lines.

1. \(\text{size} \geq 0\)
   - **Answer:** Both. “add” can only increment “size”, and “choose” does not decrement “size” when it is zero.

2. \(\text{size} \geq \text{elements.length}\)
   - **Answer:** Both. Neither method changes the length of the array, only “add” increments “size”, and if “size” is equal to “capacity”, the first statement in “add” will throw an `ArrayOutOfBoundException` and the increment will not happen.

3. for all \(i: \text{int}[i \geq 0 \& i < \text{size} \Rightarrow \text{elements}[i] \neq \text{null}]\)
   - **Answer:** Only “choose”, because it sets no array elements and only decrements the range of the quantifier. “add” does not check that the argument is non-null.

4. for all \(i,j: \text{int}[i < j \& i \geq 0 \& j < \text{size} \Rightarrow \text{elements}[i].\text{compareTo}(\text{elements}[j]) \leq 0]\)
   - **Answer:** Only “choose”, because it does not mutate the array and only decrements the range of the quantifier. “add” inserts a new element and does not compare it with already inserted elements.

5. for all \(i: \text{int}[i \geq \text{size} \& i < \text{elements.length} \Rightarrow \text{elements}[i] == \text{null}]\)
   - **Answer:** Only “add”, because it decreases the range of the quantifier, while “choose” fails to set “elements[size]” to null. If “add” is invoked with a null argument and “size” equals “elements.length”, then the first statement of “add” will throw an exception, its second statement will not be executed and the array will not be mutated, so that the invariant will be preserved.
(5 points) **Problem 7. Abstraction functions.** Consider the following class definition:

class IntSet {
    // AF: Represents a set of integers.
    // RI: "elts" is not null, "elts" contains no duplicates.
    List elts;

    IntSet( ) {
        elts = new ArrayList ( );
    }
    void add(Integer i) {
        if (!elts.contains(i))
            elts.add(i);
    }
    void contains(Integer i) {
        returns elts.contains(i);
    }
}

Map each of the following concrete representations to its abstract value, if it exists. Use the following notation: \([a, b, c]\) represents the "elts" array list, \(\{a, b, c\}\) represents the abstract mathematical set containing the distinct elements \(a, b,\) and \(c\). Write "no value" if no abstract value exists.

<table>
<thead>
<tr>
<th>Concrete Representation</th>
<th>Abstract Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3]</td>
<td>({1, 2, 3})</td>
</tr>
<tr>
<td>[1,2,2]</td>
<td>no value</td>
</tr>
<tr>
<td>[3,2,1]</td>
<td>({1, 2, 3})</td>
</tr>
<tr>
<td>[]</td>
<td>({})</td>
</tr>
<tr>
<td>null</td>
<td>no value</td>
</tr>
</tbody>
</table>

(15 points) **Problem 8. Subtyping and the Substitution Principle.** We consider a type “Counter” and potential subtypes of it, according to the Substitution Principle (not according to Java).

(1) Complete the specification of the type “Counter” below by providing the “OVERVIEW” section:

```java
public class Counter {
    // OVERVIEW: Counter is used to represent a non-negative integer.
    // Counter can be increased by one. Counter can never be decreased.

    // constructors:
    public Counter() //EFFECTS: makes this contain 0
    // methods
    public int get() //EFFECTS: returns the value of this.
    public void incr()
    // MODIFIES: this
    // EFFECTS: increments the value of this.
}
```
(2) Consider a potential subtype of “Counter”, called “Counter2”, with the following extra operations:

```java
public Counter2 () //EFFECTS: makes this contain 0
public void incr()
    // MODIFIES: this
    // EFFECTS: makes this contains twice its current value
```

Is “Counter2” a legitimate subtype of “Counter”? Justify in at most 2-3 lines.

**Answer**: “Counter2” is not a subtype of “Counter”. The postcondition of “Counter2.incr” does not imply the postcondition of “Counter.incr”. Another way of saying this: “Counter2” violates the *Methods Rule*, of the Substitution Principle, in the method “incr”.

(3) Consider another potential subtype of “Counter”, called “Counter3”, with the following extra operations:

```java
public Counter3 (int n) //EFFECTS: makes this contain integer n.
public void incr (int n)
    // MODIFIES: this.
    // EFFECTS: if n > 0, adds n to this.
```

Is “Counter3” a legitimate subtype of “Counter”? Justify in at most 2-3 lines.

**Answer**: “Counter3” is not a subtype of “Counter”. Note that “Counter3.incr” is an overloaded method name and does not override “Counter.incr”, because the two methods do not have the same input parameters. Thus, “Counter3” does not violate the *Methods Rule* of the Substitution Principle. However, if we consider the specification in part (1) above, it requires that “Counter” must hold a non-negative integer, whereas “Counter3” can be constructed with a negative integer “n”, thus violating the *Properties Rule* of the Substitution Principle.

(10 points) **Problem 9. Subtyping and the Substitution Principle.** We say that an interface A is a *true* supertype of an interface B, and B is a *true* subtype of A, if A is a supertype of B according to the Substitution Principle.

In the following table, the first column is the specification of a method from an interface A, and the second column is the specification of a corresponding method from an interface B. In the third column, write YES if interface B could potentially be a true subtype of interface A, and write NO if interface B could not be a true subtype of interface A. Each row of the table is independent of the other rows. You may wish to add a short explanation (no more than 2 lines) for each of your answers.

<table>
<thead>
<tr>
<th>Method from A</th>
<th>Method from B</th>
<th>Could B be a true subtype of A?</th>
</tr>
</thead>
<tbody>
<tr>
<td>void insert1 (int x) // MODIFIES: this // EFFECTS: adds x to the contents of this.</td>
<td>void insert1 (int x) // MODIFIES: this // EFFECTS: if x is odd, adds it to this.</td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td>void insert2 (int x) // MODIFIES: this // EFFECTS: adds x to the contents of this.</td>
<td>void insert2 (int x) // MODIFIES: this // EFFECTS: adds x to the contents of this and prints x to the screen.</td>
<td><strong>YES</strong></td>
</tr>
<tr>
<td>void addZero( ) // REQUIRES: this is not empty. // MODIFIES: this. // EFFECTS: adds 0 to this.</td>
<td>void addZero( ) // MODIFIES: this // EFFECTS: adds 0 to this.</td>
<td><strong>YES</strong></td>
</tr>
</tbody>
</table>
public class Buggy {
    // returns: the first index, between left and right inclusive, of n in array
    // throws: RuntimeException if n does not appear in a[left..right]
    private static int indexOf(int[] a, int n, int left, int right) {
        // ... recursive implementation omitted ...
    }
    public static int indexOf(int[] a, int n) {
        return indexOf(a, n, 0, a.length-1);
    }
    public static void main(String argv[]) {
        int[] a = { 0, 5, 3, 2, 7, 6, 9, 1, 4, 8 };  
        System.out.println("indexOf(0): " + indexOf(a,0));
        System.out.println("indexOf(1): " + indexOf(a,1));
    }
}

Suppose the program prints the index of 0, as expected, and then throws an exception indicating there is a bug somewhere in the program. Give two hypotheses for the source of the error, i.e. two possible bugs causing the error. Briefly discuss both of your hypotheses: How you would test each of your hypothesis (i.e. whether it is indeed the source of the error) and what you would learn from your investigation.

Answer: We give 3 possible hypotheses. We give you full credit for two.

(1) One hypothesis is that the second operation always fails, perhaps because of caching in the implementation. This can be tested by trying to look up the same value twice. If the second lookup succeeds, then the hypothesis is refuted.

(2) Another hypothesis is that the array needs to be sorted. This can be tested by trying the same lookups, i.e. 0 and 1, but changing a as follows:

    int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    If the lookup of 1 still fails, then the hypothesis is refuted.

(3) Another hypothesis is that lookups anywhere except the first element fail. This can be tested by trying the same lookups (either in the same order or a different order), but changing a as follows:

    int[] a = { 1, 0 };

    If the lookup of 0 still works, then the hypothesis is refuted.