

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
6.170 LABORATORY IN SOFTWARE ENGINEERING  
FALL 2000

Quiz, Part 1  
SOLUTIONS  
October 31, 2000

There are 3 sections (labeled A through C), forming 7 pages on 4 sheets. There is a separate answer sheet at the end. Please check your copy of the quiz before you start to make sure it is complete.

Only the answer sheet will be graded. You may write on this handout, but it will be discarded, so you should make sure to transcribe all your answers on to the answer sheet itself.

You have 55 minutes and should attempt to answer all questions. Note that the questions are not equally weighted. The answer sheet shows the number of points allotted to each question.

**Remember to write your name on your answer sheet!**

## A. Quickies

Answer yes or no by placing a 1 (true) or 0 (false) in the appropriate box on the answer sheet.

### Java

1. A constructor of class *C* cannot take an argument of class *C*. *False. Copy often implemented as a constructor. Also recursive types (eg, cons for list).*
2. A class may have at most one constructor. *False.*
3. An object's type cannot change at runtime. *True; downcasts just perform a check.*
4. The constructor of a class cannot return objects of a subclass. *True; that's why factory methods are coded as static methods.*
5. Downcasts may change the state of mutable objects. *False.*
6. A subclass may override but not eliminate methods. *True.*
7. Interfaces can have constructors. *False; this is one of the problems of using interfaces for data abstraction.*
8. A class may extend more than one class. *False. Java does not have multiple inheritance of classes, but it does have multiple inheritance of interfaces (sometimes called "specification inheritance").*
9. A class may implement more than one interface. *True.*
10. Every object has a *hashCode* method. *True; inherited from Object.*
11. Primitive values may not be stored in arrays. *False.*
12. Primitive values may not be stored in *Vectors*. *True; the add method of Vector takes an argument of type Object.*
13. For immutable objects, *equals* and *==* are equivalent. *False; consider String, eg.*
14. *null* can be regarded as an object of class *Object*. *False; can't call method on null.*
15. Some iterators allow elements to be removed during iteration. *True; see Java API.*
16. A call to a method that throws a checked exception must appear in a *try-catch*. *False; can propagate.*
17. Garbage collection may cause a reference to become *null* unexpectedly. *False. Garbage collection eliminates memory allocation errors.*
18. *Arrays* are more fundamental to the Java language than *Vectors*. *True, since the Vector class is part of the standard library, but arrays are part of the language itself. That's why you can declare an array of a particular element type. We did not grade this question, however, since we decided that it was ambiguous. From the point of view of the user of the language, Vectors (or ArrayLists) are arguably more fundamental since they are extendable and therefore more useful.*
19. Assigning *null* to an argument just before returning has no effect on callers of a method. *True; it's call by value.*
20. If *x* is *null*, the call *o.m(x)* must throw an exception. *False.*
21. If *o* is *null*, the call *o.m(x)* must throw an exception. *True.*
22. If the compiler admits the expression *o.m(x)*, *o* cannot be bound at runtime to an object that does not have method *m*. *True; that's what compile-time typing gives you.*

- 23. Unchecked exceptions cannot be user-defined. *False.*
- 24. A constructor cannot have side effects. *False.*

#### *Programming style*

- 25. It is good practice to catch every unchecked exception that might be raised. *False; virtual machine errors, eg, should not have handlers since there is nothing useful that can be done.*
- 26. Java interfaces should be avoided, since they damage performance. *False.*
- 27. Widespread use of *instanceof* is evidence of clumsy structuring. *True.*
- 28. Checked exceptions are always preferable to unchecked exceptions. *False.*
- 29. Looping over an index is usually better than using an iterator. *False.*
- 30. Only observers should throw exceptions. *False.*
- 31. A type should provide a few, orthogonal methods instead of many specialized ones. *True.*
- 32. In general, the deeper the inheritance hierarchy, the better. *False.*
- 33. Exceptions should never be thrown during the normal execution of a program. *False, as taught in this course, although some experts disagree.*

#### *Classifying operations and types*

- 34. An operation that is an observer may not mutate the representation. *False; beneficial side effects.*
- 35. A mutator operation performs a mutation on every call. *False.*
- 36. An object with a field that references a mutable object must be mutable. *False.*
- 37. Immutable types generally admit more sharing of substructures. *True.*
- 38. A call to a producer always result in copying being performed. *False; producers of immutable types use sharing to avoid copying.*
- 39. Exceptions cannot be mutable. *False; exceptions are user-definable objects.*
- 40. Mutable types don't need constructors. *False; all abstract data types need a constructor.*

## B. Abstract types, Representations & Invariants

You have been asked to design a *Set* datatype, and have devised this initial specification:

```
class Set {
  Set ();
  // effects: constructs a new, empty set

  void add (Object o);
  // effects: inserts the object o into this set

  void delete (Object o);
  // effects: removes from this set every element equal to o

  boolean member (Object o);
  // effects: returns true iff there is an element in this set equal to o

  int size ();
  // effects: returns the size of this set
}
```

1 Your client has requested 3 additional operations:

- (a) `Set (Set s);`  
// effects: constructs a new set containing the same elements as this set
- (b) `void removeAll (Set s);`  
// effects: removes from this set every element equal to an element of s
- (c) `boolean isEmpty ();`  
// effects: returns true iff this set is empty

You want to help, but you'd rather keep the specification small and simple. Which of the 3 are you least inclined to add? Select one of (a), (b) or (c).

*(c). The client can easily compare the size to zero, but omitting either of the other methods will require the client to write a loop.*

2 You are considering changing the specification of `delete` to:

```
void delete (Object o);
// effects:
// if o is a member of this set, removes it
// otherwise throws the exception NotFoundException
```

Should *NotFoundException* be checked? Mark a 0 (unchecked) or 1 (checked) in the box.

*Unchecked, since otherwise awkward for client that uses member to ensure that delete is never called in this case.*

3 It turns out that only sets containing fewer than 100 elements are needed. You plan to change the specification accordingly to allow more efficient implementations, but in such a way that you don't upset your client too much. Where should you make the change? Select one of (a), (b), (c) or (d).

- (a) precondition of *add*

- (b) precondition of *delete*
- (c) postcondition of *add*
- (d) postcondition of *delete*

(c). The *add* method must refuse the element that would cause the bound to be exceeded.

You are considering four possible representations of the original, unbounded *Set*:

- (a) A hashtable, using the standard Java library type
- (b) A vector, using the standard Java library type
- (c) An array, using Java's built in array construct
- (d) A sorted binary tree, which you will code yourself

For each of the questions 4 thru 6, select one of (a), (b), (c) or (d).

- 4 Which will support *member* running in constant time? (a). (b) and (c) are not inconsistent with *member* running in constant time, since they could be augmented by a hashtable, but they do not support it.
- 5 Which will *not* allow *add* to be implemented in constant time? (d).
- 6 Which is likely to be hardest to code? (d).

You decide to use the array representation, with perhaps some additional fields, and are considering including the following clauses as part of your rep invariant, in which *r* denotes the representation object, which has fields *size* and *arr*:

- (a)  $r.size = \min \{i : 0..r.arr.length \mid r.arr[i] = \text{null}\}$
- (b)  $\text{all } i, j: 0..r.size-1 \mid r.arr[i] = r.arr[j] \Rightarrow i = j$
- (c)  $\text{all } i, j: 0..r.size-1 \mid i \leq j \Rightarrow r.arr[i].lessThanOrEqual(r.arr[j])$
- (d)  $\text{all } i, j: 0..r.size-1 \mid r.arr[i].equals(r.arr[j]) \Rightarrow i = j$

Questions 7 to 13 inclusive refer to these invariants.

7 Which one implies another? Put one of (a) thru (d) in each of the two boxes.  $d \rightarrow b$ . If two references are to the same object, clearly the object referenced by one is equal (using an *equals* method) to the object referenced by the other. Since the equality tests appear on the left-hand side of the implication, the invariant that uses *equals* implies the invariant that uses *==*, rather than the converse. If the array contains no duplicates that are value equal, it certainly cannot contain two references to the same object!

For each of the questions 8 thru 12, select one of (a), (b), (c) or (d).

- 8 Which makes *add* implementable in constant time? (a): requires no check for presence, or any sorting.
- 9 Which must *not* be used if *add* is to be implemented in constant time? (c): requires finding appropriate index. (b) and (d) forbid duplicates, but could augment the rep with a hashset to check first.

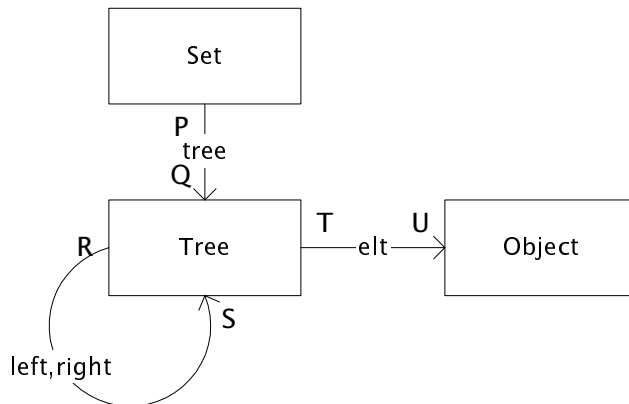
**10** Which is not appropriate because a *Set* may contain immutable objects? (b): *should use equals method, not object equality. Because of a typo in the original version of the test, we did not grade this question.*

**11** Which requires a change to the specification? (c), *because Object does not provide lessThanOrEqual.*

**12** Which allows null references to be stored as elements? (b), *since (a) uses the first null to bound the array, and (c) and (d) use methods on the elements.*

**13** Which two may suffer from rep exposure? Select two of (a), (b), (c) or (d). (c) and (d), *since both involve methods of the element type whose behaviour might change if the element is mutated.*

Later, you discover that a tree representation is preferable. To design it, you start with a code object model that has no multiplicity markings:



You now add multiplicity markings to express crucial properties. For each of the properties 14 thru 17, select *one* site from *P* thru *U*, at which the multiplicity marking ? should be placed.

**14** Subtrees are not shared. *R*

**15** Null references can be stored. *U*

**16** The empty set is representable. *Q*

**17** Finite sets are representable. *S*; if there is a ! rather than a ? here, every tree must have subtrees, so must be infinite. This is the only answer that makes any sense, but it's not actually quite right. One could represent finite sets by having a tree that contains itself rather than a null reference to a tree.

**18** Which of the markings that you selected in 14-17 cannot be expressed as a rep invariant? Select one of 14, 15, 16 or 17.

*14, since the rep invariant is about a single set, and 14 rules out sharing between trees that belong to different sets. The marking for 17 is expressible as a rep invariant, but the property itself is not. If you chose 17, you also received credit, since the question should have been clearer.*

Which of the following additional properties can be expressed by adding multiplicity or mutability markings to the object model? For each of 19-21, mark a 0 (for no) or 1 (for yes) in the appropriate box.

**19** An element may appear at most once as a leaf of a given tree. *No; can only express that it can't appear as leaf of two subtrees, but this would rule out an element being in two sets.*

**20** Trees are balanced. *No.*

**21** Entire trees may not be shared between set objects. *Yes.*

### C. Module Dependences

You are reviewing a draft design of a banking system that includes the following class outlines.

```
class Bank {
    Bank ();
    void addAccount (Account a);
    Account lookupAccount (String name);
    void post (Account a, Transaction t);
    Collection getAllTrans (Account a, Date from, Date to);
}

class Account {
    Account (String name);
    String getName ();
    void post (Transaction t);
    Collection getAllTrans (Date from, Date to);
    int balance ();
}

class Transaction {
    Transaction (Date d, int amt);
    Date getDate ();
    int getAmount ();
}

class Date {
    Date (int year, int month, int day);
    int compareTo (Date other);
}
```

To evaluate the quality of the design, you construct a module dependency diagram. The designers have not made things easy for you: there are no specs, so you have to guess what the methods do from their names. As always when constructing dependences to evaluate design, you will need to think about how methods will be implemented, and which dependences will be *necessary*.

- 1 Construct a diagram for five classes – *Bank*, *Account*, *Trans*, *Date* and *String* – by completing the matrix, writing *S* for a strong dependence, *W* for a weak dependence and a dash for no dependence. Self-dependences are shaded out: you should ignore them. Also ignore dependences due solely to inheritance. For example, if you think there must be a strong dependence of *Bank* on *String*, but no others, you would write *S* in the entry in the row labelled *Bank* and column labelled *String*, and a dash in every other unshaded entry.

*Bank has strong dependences on Account and String (because it must compare strings to do lookUp, and string comparison cannot be done using ==), and weak dependences on Transaction and Date. If Account did not provide the getAllTrans method, Bank would have strong dependences on Transaction and Date. Account has strong dependences on Transaction and Date (because the implementation of getAllTrans must call compareTo), but only a weak dependence on String. Transaction has a weak dependence on Date, because of the date field. There are no other necessary dependences. This is quite challenging, but it is exactly what needs to be done to evaluate a design from its specification before code has been written.*

You are considering using a *List* class to link banks to accounts and accounts to transactions. Suppose the *List* class has only these methods:

```
class List {
    List ();
```

```
int addObject (Object o);  
Object getObjectAt (int index);  
}
```

- 2 Construct the module dependency diagram for the new system, this time considering the classes *Bank*, *Account*, *Trans*, *List* and *Object* – that is, all classes except for *Date* and *String*. Ignore dependences due solely to inheritance.

*Bank and Account have strong dependences on List. List itself only has a weak dependence on Object, because it offers no method to find elements using equality. No other class has a dependence on Object.*