

**Massachusetts Institute of Technology
6.170 Laboratory in Software Engineering**

Fall 2004

Quiz 1 Solutions

Monday, March 1, 2004

Name: _____

Athena User Name: _____

Section (check one):

- | | | |
|--|---|---|
| <input type="checkbox"/> 1. Matt Notowidigdo | <input type="checkbox"/> 4. Brian Dunagan | <input type="checkbox"/> 7. Gregory Marton |
| <input type="checkbox"/> 2. Magdalena Balazinska | <input type="checkbox"/> 5. Cliff Frey | <input type="checkbox"/> 8. Paul Pham |
| <input type="checkbox"/> 3. Michael Bolin | <input type="checkbox"/> 6. Ryan Jazayeri | <input type="checkbox"/> 9. Asfandyar Qureshi |

Instructions

This quiz is closed book, closed notes. You have **50 minutes** to complete it. It contains 18 questions in 12 pages (including this page), totaling 100 points. Before you start, please check your copy to make sure that it is complete. Turn in all pages, together, when you finish.

Write your recitation section number and your initials at the top of each page.

Please write neatly; no credit will be given if we cannot read what you write.

True/false questions. Circle T if the statement is true, F if not.
[5 pts each]

1. T F If class A overrides Object's toString(), there is no way for a different class to call Object's toString() on instances of A.

True. Once A has overridden toString(), the original toString() method can only be called from within A, using super.toString(). Other classes can't call it on A at all.

2. T F The modifies clause must list any local variables modified by the method.

False. In fact, the method's local variables should not be listed in the modifies clause, since they're private to the method's implementation

3. T F Adding a requires clause makes a specification stronger.

False. Adding a requires clause strengthens the precondition, which makes the specification weaker.

4. T F "requires: array is sorted" means the method doesn't check whether the array is sorted.

False. Methods are free to check whether their preconditions hold. "requires: array is sorted" only means that the method's effect is unspecified if the array isn't sorted.

5. [12 pts] Which of the following methods/constructors cannot possibly expose the rep? (check all that apply)

public int solveEquations(int x, int y, int z)
Parameters and return value are immutable.

public String[] decode(boolean slowly)
Return value is a mutable array, which might be part of the rep.

private Date myBirthday()
Private methods can't expose the rep.

public String toString()
Return value is immutable.

public Iterator elements()
Iterator.remove() might affect the rep.

public Deck(List cards)
Parameter is a mutable object, which might be saved in the rep.

The questions on the next three pages use the code below. First, suppose the standard Float class, which represents a floating point number as an object, is implemented as follows.

```
public class Float {
    private float f;

    public Float(float f) {
        this.f = f;
    }

    public float floatValue() {
        return f;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Float))
            return false;

        float f1 = this.floatValue();
        float f2 = ((Float) o).floatValue();
        return (f1 == f2);
    }
    // more methods here that don't interest us now
}
```

Alyssa P. Hacker decides to improve on Float by making a subclass TolerantFloat that tolerates small errors when deciding whether two float values are equal. Here's her code:

```
public class TolerantFloat extends Float {
    public static final float TOLERANCE = 0.01;

    public TolerantFloat(float f) {
        super (f);
    }

    public boolean equals(Object o) {
        if (!(o instanceof Float))
            return false;

        float f1 = this.floatValue();
        float f2 = ((Float) o).floatValue();
        return (Math.abs(f1 - f2) <= TOLERANCE);
    }
}
```

6. [5 pts] What will be the effect of the following code?

```
TolerantFloat tf1 = new TolerantFloat (5.0);
TolerantFloat tf2 = new TolerantFloat (5.0000001);
Float f1 = (Float) tf1;
Float f2 = (Float) tf2;

if (tf1.equals(tf2)) {
    if (f1.equals(f2))
        System.out.println ("exactly same");
    else
        System.out.println ("almost same");
} else {
    System.out.println ("different");
}
```

Check only one:

___ throws NullPointerException

___ throws ClassCastException

X prints "exactly same"

Dynamic dispatch means TolerantFloat.equals() is called both times.

___ prints "almost same"

___ prints "different"

7. [9 pts] Does `TolerantFloat.equals()` satisfy each of the following properties required by the Object contract?
- If so, just write **Yes**.
 - If not, give a **counterexample**. To save writing, you can write F for `Float` and TF for `TolerantFloat`.

(a) `equals()` is reflexive

Yes: $x.equals(x)$ for all `Floats` and `TolerantFloats` x .

(b) `equals()` is symmetric

No:

*`Float f = new Float(1);`
`TolerantFloat tf = new TolerantFloat(1.01);`*

*`tf.equals(f) == true`
`but f.equals(tf) == false`*

(c) `equals()` is transitive

No:

*`TolerantFloat tf1 = new TolerantFloat(1);`
`TolerantFloat tf2 = new TolerantFloat(1.01);`
`TolerantFloat tf3 = new TolerantFloat(1.02);`*

*`tf1.equals(tf2) == true`
`tf2.equals(tf3) == true`
`but tf1.equals(tf3) == false`*

Now, Alyssa adds a hashCode() method to Float and TolerantFloat. She knows that in Java, casting a float value to an int truncates the fractional part (removing all the digits after the decimal point). So she writes this code:

```

public class Float {
    ...
    public int hashCode() {
(a)         return (int) floatValue();
    }
}

public class TolerantFloat extends Float {
    ...
    public int hashCode() {
(b)         return (int) floatValue();
    }
}

```

8. [3 pts] Ben Bitdiddle correctly complains that TolerantFloat's hashCode() doesn't satisfy the Object contract. Give an example that proves Ben's point. (Again, you can write F for Float and TF for TolerantFloat.)

```

TolerantFloat tf1 = new TolerantFloat(4.9);
TolerantFloat tf2 = new TolerantFloat(5.0);

```

```

tf1.equals(tf2) == true
but tf1.hashCode() == 4
   tf2.hashCode() == 5

```

9. [3 pts] Louis Reasoner goes further than Ben: "There's no way you can implement hashCode() for Float and TolerantFloat that will satisfy the Object contract." Louis is wrong. In the space below, write replacements for lines (a) and (b) in the code above so that hashCode() satisfies its specification.

(a) *return 1;*

(b) *return 1;*

Any int constant would work, as long as both Float and TolerantFloat used the same one.

The questions on this page use the Account data type shown below.

```
public class Account {
    private String name;
    private List transactions;
    public Account(String n) {
        name = n;
        transactions = new ArrayList();
    }
    public Account(Account a) {
        name = a.name;
        transactions = new ArrayList(a.transactions);
    }
    public boolean post(Trans t) {
        transactions.add(t);
        return true;
    }
}
```

10. [3 pts] Classify each operation as a **creator**, **mutator**, **producer**, or **observer**.

Account (String n) is a: *creator*

Account (Account a) is a: *producer*

boolean post (Trans t) is a: *mutator*

11. [6 pts] Assume that Account inherits no methods at all from Object, so its only methods are the ones you see above. Now suppose we run the following code:

```
Account a = new Account ("Frodo");
Account b = new Account ("Sam");
Account c = a;
Account d = new Account (a);
```

Which of the following are *referentially* equivalent to a? (Circle all that apply)

c only.

Which of the following are *behaviorally* equivalent to a? (Circle all that apply)

b, c, d. Account objects can't be distinguished by the methods available.

Which of the following are *observationally* equivalent to a? (Circle all that apply)

b, c, d. Same reason.

Louis Reasoner has made the following statements, but unfortunately for him they're all false. Explain why they're false. (A good answer won't be long. Use 20 words or less.)

12. [6 pts] “Mutable classes don't need to worry about rep exposure, because clients can already modify the rep by calling mutators.”

Even mutable classes have a rep invariant that they need to preserve. Rep exposure threatens that rep invariant.

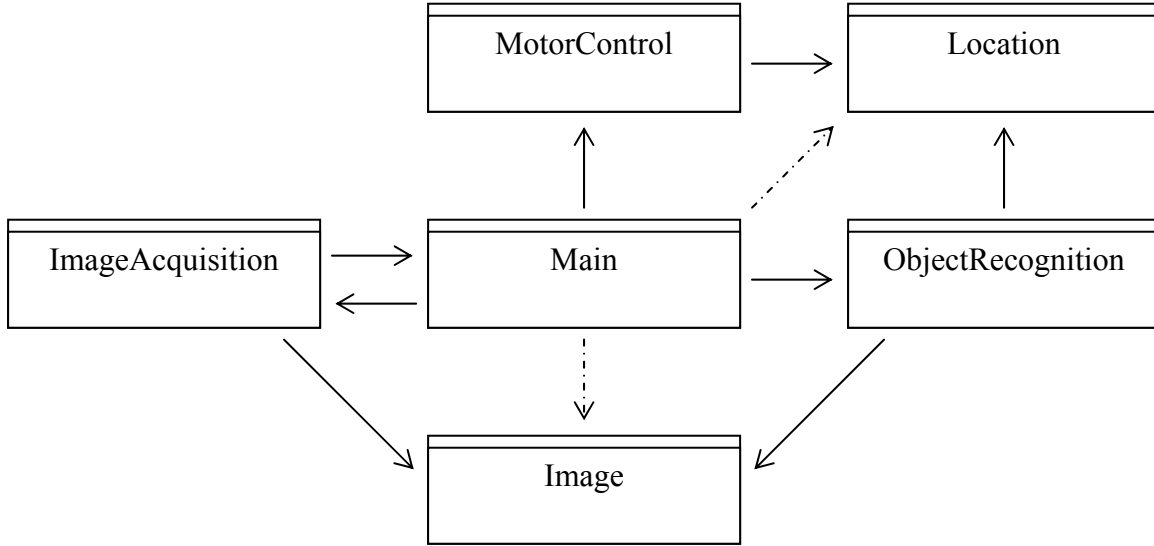
13. [6 pts] “Observers should never modify the rep.”

An observer can modify the rep with a benevolent side effect – a side effect that doesn't change the abstract value represented by the rep.

14. [6 pts] “The abstraction function must be defined on all values of the rep space.”

The abstraction function only needs to be defined on rep values that satisfy the rep invariant.

Use this module dependency diagram to answer the questions at the bottom of this page.



15. [3 pts] If the implementation of a method in Image changes, but all specifications in the system remain the same, which other classes might need to be updated?

Check all that apply:

_____ ImageAcquisition

_____ Main

_____ ObjectRecognition

_____ MotorControl

_____ Location

 X none of the above

16. [3 pts] If the specification of a method in Image changes, but all other specifications in the system remain the same, which other classes might need to be updated?

Check all that apply:

 X ImageAcquisition

_____ Main

 X ObjectRecognition

_____ MotorControl

_____ Location

_____ none of the above

17. [8 pts] The mean() method computes the average of an array of numbers. Here are three proposed implementations. (The differences are highlighted in boldface.)

A. GOOD

```
static double mean (int[] a, int n) {  
    double sum = 0;  
    for (int i = 0; i < n; i++)  
        sum += a[i];  
    return sum / n;  
}
```

B. GOOD

```
static double mean (int[] a, int n) {  
    if (a == null) return 0;  
    double sum = 0;  
    for (int i = 0; i < n; i++)  
        sum += a[i];  
    return sum / n;  
}
```

C. BAD

```
static double mean (int[] a, int n) {  
    return 0;  
}
```

Write a complete, declarative specification for mean() that is satisfied by implementations A and B but not C. (Don't assume any default meanings for requires, modifies, or effects. Don't leave any of them blank.)

requires: *a != null and 0 < n <= a.length*

modifies: *nothing*

effects: *returns the mean of the first n numbers in a*

18. [7 pts] This question uses the following abstract data type, which represents a face-down deck of cards that can be turned up one at a time onto a face-up discard pile.

```
public class Dealer {
    private Deck faceDown;
    private Deck faceUp;
    private Card card;

    public Dealer() {
        faceDown = new Deck();
        faceDown.shuffle();
        faceUp = new Deck();
        faceUp.removeAllCards();
    }

    public void turnUpNextCard() {
        Iterator dealtCards = faceDown.dealCards(1);
        if (dealtCards.hasNext()) {
            card = (Card)dealtCards.next();
            faceUp.addCard(card);
        }
    }

    public Card getCardShowing() {
        return card;
    }
}
```

Write a complete rep invariant for Dealer, using the space below. (If you need the specification for Deck to understand Dealer or write your rep invariant, you can find it on the next page.)

faceDown != null && faceUp != null
faceDown and faceUp have no cards in common
faceDown + faceUp = all 52 possible cards
card = null if faceUp is empty
card = bottom card of faceUp if faceUp is nonempty

Here is the specification for Deck, in case you need it to understand Dealer. Some new methods have been added compared to the Deck you used in PS1, but the original methods still behave the same way.

overview: Deck represents a mutable sequence of playing cards.

```
public class Deck {  
  
    public Deck();  
        effects: creates a deck containing 52 cards in sorted order  
  
    public void addCard(Card c);  
        modifies: this  
        effects: no effect if c is null or a duplicate of a card already in this deck;  
        otherwise adds c to the bottom of this deck.  
  
    public void removeCard(Card c);  
        modifies: this  
        effects: no effect if c is null or does not exist in this deck;  
        otherwise removes c from this deck.  
  
    public void removeAllCards();  
        modifies: this  
        effects: removes all cards from this deck.  
  
    public boolean containsCard(Card c);  
        effects: returns true if and only if c is in this deck.  
  
    public void shuffle();  
        modifies: this  
        effects: randomly permutes the order of the cards in this deck  
  
    public Iterator dealCards(int n);  
        modifies: this  
        effects: if n <= 0, returns an empty iterator and has no effect on this deck;  
        if n >= number of cards in this deck, removes all cards from this deck  
        and returns them in an iterator;  
        otherwise removes n cards from the top of the deck and returns them in  
        an iterator.  
  
    public Iterator listCards();  
        effects: returns iterator listing the cards of this deck, ordered from top to bottom.  
  
    public int size ();  
        effects: returns the number of cards in this deck.  
  
}
```