



## True/False [1 point each]

1. Mutable objects cannot be interned. **True.**
2. Decentralized organization permits meetings to proceed effectively without a leader. **False.**
3. A good design does not rule out any future extension to a project. **False.**
4. It is typically preferable to allow a project to slip gradually rather than enduring a single large slip in the schedule. **False**
5. The Composite pattern is useful for operations on hierarchical data structures. **True**
6. When examining a data structure using the Procedural pattern, the examining class must do its own type dispatching on the possible types of the data structure. **True.** (*By contrast, the Interpreter pattern requires each data type to have a method per operation.*) See sidebar 15.10 on page 393 of the text.
7. A Java interface should include an abstraction function. **False.** *Interfaces don't define representations.*
8. Introducing interfaces can increase the total number of dependences. **True**
9. Introducing interfaces can help to eliminate dependences. **True**
10. A weak dependence of class A on class B implies an arrow from class A to class B in the code object model. **False**
11. A strong dependence of class A on class B implies an arrow from class A to class B in the code object model. **False**

For questions 12–15, consider the following code fragment:

```
/** An IntSet is a mutable set of integers.
    Examples include: { } , { 1, 2, 3 }, { 4, 5, 6, 7, 100 }
 */
class IntSet {
    REPTYPE nums;

    // RI(c): ...
    // AF(c): { i | i is in nums }
}
```

Note that questions 12–14 use the term “sensible”, while question 15 uses the term “necessary”.

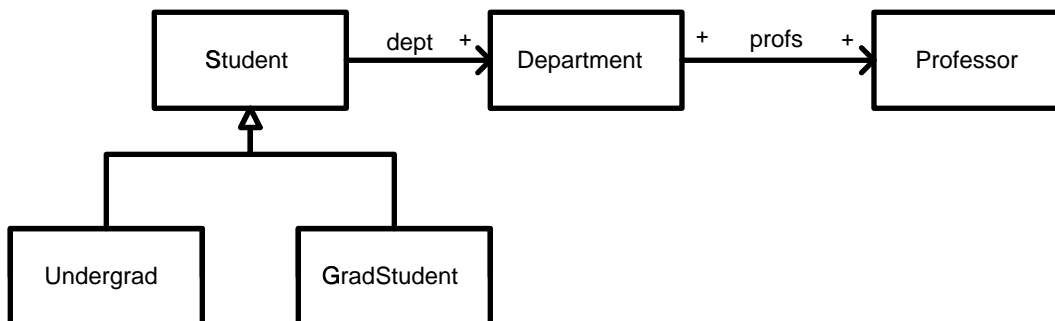
12. If REPTYPE is HashSet, “No duplicate values in c.nums” would be a *sensible* clause to include in the representation invariant. **False. The HashSet abstraction already implies that values will not be duplicated.**
13. If REPTYPE is HashSet, “o in c.nums  $\Rightarrow$  (o instanceof Integer)” would be a *sensible* clause to include in the representation invariant: **True. HashSets can contain arbitrary object types, and therefore it is sensible to constrain the object type in this representation.**
14. If REPTYPE is Integer[], “o in c.nums  $\Rightarrow$  (o instanceof Integer)” would be a *sensible* clause to include in the representation invariant. **False. The type system ensures that o is an Integer, because it is in an Integer array.**
15. If REPTYPE is Integer[], “No duplicate values in c.nums” would be a *necessary* clause to include in the representation invariant. **False. One could write an IntSet implementation that allowed for duplicated values in the representation during execution.**

Write your answers to these questions on the answer page, not on this page.

---

16. The representation of an ADT is exposed if one of its methods returns a mutable object. **False. Returning a mutable object in the rep does expose the rep. See page 111 of the book.**
17. A code object model describes the state of the heap at runtime. **True**
18. A snapshot describes the state of the heap at runtime. **True**
19. In a problem object model, the nodes represent classes and the edges represent relations or subtyping. **False**
20. “Requirement specification 50% completed” is a good milestone. **False.**
21. “Module A coding completed and unit-tested” is a good milestone. **True.**
22. “Project design completed and reviewed” is a good milestone. **True.**
23. “Module A unit testing completed” is a good milestone. **True.**
24. “Module A specification completed” is a good milestone. **True.**

Questions 25–28 relate to the object model below, which describes a university (though not necessarily MIT).



25. Every student is in some department. **True**
26. Students may switch departments. **True**
27. A student cannot switch from undergrad to grad status. **False**
28. Every department has students. **False**

## Multiple choice [2 points each]

29. A Factory can:

- (a) return a subtype of a class where a supertype is expected
- (b) return pre-existing objects rather than new ones
- (c) encapsulate the creation of an object or set of objects
- (d) (a) and (b)
- (e) (a) and (c)
- (f) (b) and (c)
- (g) (a), (b), and (c)

**G: all of the above.**

30. Which of these patterns does not control sharing?

- (a) Flyweight
- (b) Interning
- (c) Prototype
- (d) Singleton

**C. Prototype is not designed to control access to other instances.**

31. A Singleton class has:

- (a) a public constructor and a public accessor for the single instance
- (b) a private constructor and a public accessor for the single instance
- (c) a public constructor that throws an exception if called more than once
- (d) a private constructor that throws an exception if called more than once

**B. The accessor creates and returns the only instance of the class.**

32. A Proxy:

- (a) changes the interface to an underlying class
- (b) extends the functionality of an underlying class
- (c) both changes the interface and extends the functionality
- (d) neither changes the interface nor extends the functionality

**D**

33. Subjects and Observers/Listeners appear in:

- (a) the push model
- (b) the pull model
- (c) the publish/subscribe model
- (d) all of the above
- (e) none of the above

**D. See book section 15.7 and sidebar 15.12.**

34. In a well-managed large project typically the least time is spent on

- (a) planning
- (b) coding
- (c) component testing
- (d) integration testing

**B**

35. What are the three properties (from the list below) that a loop invariant must satisfy? (Write down three answers, in alphabetic order, in the three boxes of the answer sheet.)

Assume that the loop is of the form

$P \{ \text{while } (b) \ S \} Q$

where  $P$  is the precondition and  $Q$  is the postcondition; let  $I$  be the loop invariant.

- (a)  $P \Rightarrow I$
- (b)  $P \Rightarrow Q$
- (c)  $I \ \&\& \ !b \Rightarrow Q$
- (d)  $I \Rightarrow Q$
- (e)  $I \{ S \} Q$
- (f)  $I \ \&\& \ b \{ S \} I$
- (g)  $I \ \&\& \ P \{ S \} Q$
- (h)  $I \{ S \} Q \Rightarrow I$
- (i)  $I \{ S \} \ !b \Rightarrow Q$
- (j)  $Q \Rightarrow !P$
- (k)  $Q \Rightarrow !b$
- (l)  $Q \Rightarrow I$

*The loop invariant  $I$  must satisfy:*

- (a)  $P \Rightarrow I$
- (c)  $I \ \&\& \ !b \Rightarrow Q$
- (f)  $I \ \&\& \ b \{ S \} I$

**Short answer [approximately 6 points each]**

36. (4 points) What is the weakest precondition for statement composition, i.e.,  $wp((S1 ; S2), Q)$ ?  
 $wp(S1, wp(S2, Q))$

37. (6 points) The *strongest postcondition*  $sp$  is defined analogously to the weakest precondition  $wp$ . The strongest postcondition is useful for forward reasoning about code. What is  $sp((x = y + 1; y = 2 * x), (y > 10 \ \&\& \ x > 0))$ ?

$x > 11 \ \&\& \ y = 2x$

***The postcondition  $x > 11 \ \&\& \ y > 22$  is weaker; it is not the strongest postcondition.***

38. (6 points) Give two guidelines for at what time a prototype should be discarded.

***The previously set deadline for discarding the prototype is reached.***

***The specific questions which the prototype was intended to answer have been answered.***

39. (5 points) Which sorts of projects is the waterfall model good for?

*Projects where customer and vendor both understand the requirements; for example, incremental releases or customizations.*

40. (6 points) Give three reasons why adding people to a project is usually inefficient.

*The new members are not initially productive because of their learning curve.*

*The old members become less productive because they are helping the new members.*

*Communication overhead increases as members are added to a project.*

*In a larger project, there is more likely to be repeated work.*

41. (7 points) Write a decrementing function (as a legal Java expression) and a loop invariant (which may be either formal or informal) for the following code fragment:

Precondition: `sbuff` is a character array possibly containing a 0 character (`'\0'`).

Postcondition: `s` is a `String` which is made up of the characters from `sbuff` up to, but not including, the first 0 character in `sbuff`, if any.

```
String s = "";
for (int i=0; sbuff[i] != '\0' && i < sbuff.length; i++) {
    s += sbuff[i];
}
```

You may find it more convenient to consider the following equivalent code; the answer is the same regardless of which version of the code you use.

```
String s = "";
int i=0;
while (sbuff[i] != '\0' && i < sbuff.length) {
    s += sbuff[i];
    i++;
}
```

(a) Loop invariant:

- `s.length() == i`, *and*
- *the first  $i$  characters of `sbuff` are equal to the first  $i$  characters of `s`. Formally,*  
 $0 \leq j < i \Rightarrow s.get(j) == sbuff[j]$ .

*The point of a loop invariant is not merely to state facts that are true about the loop body; it is to state properties that are useful in proving the loop correct.*

(b) Decrementing function:

`(sbuff.length - i)`

*Equivalently:* `(sbuff.length - s.length())`

42. Consider a collection class C with a representation made up of two integer arrays:

```
class C {
    int[] a;
    int[] b;
    ...
}
```

One concrete value is this one:

$$\begin{aligned} a &= [-2, 4, 8, 3] \\ b &= [1, 4, 10, 6] \end{aligned}$$

Hint: think about pairs, such as  $\langle a[i], b[i] \rangle$ .

- (a) Assume that C can represent any set. Write a representation invariant and an abstraction function for an implementation in which the concrete value represents the set

$$\{-2, -1, 0, 1, 3, 4, 5, 6, 8, 9, 10\}.$$

(Hint: this is similar to something you saw on a problem set.) Your answer may be formal or informal, but must be clear.

**representation invariant:**

$$\begin{aligned} a \neq null \ \&\& \ b \neq null \\ a.length &= b.length \\ \forall i. 0 \leq i < a.length &\Rightarrow a[i] \leq b[i] \end{aligned}$$

**abstraction function:**  $\{e \mid \exists i. 0 \leq i < a.length \ \&\& \ a[i] \leq e \leq b[i]\}$

- (b) Assume that C can represent any multiset. Write a representation invariant and an abstraction function for an implementation in which the concrete value represents the multiset (bag)

$$\{-2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 3, 3, 3, 3, 3\}.$$

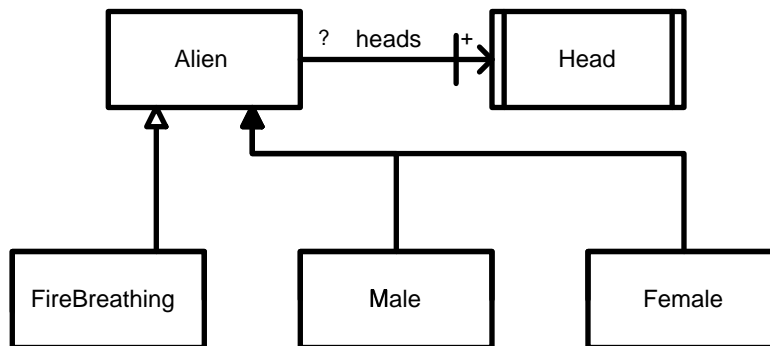
Your answer may be formal or informal, but must be clear.

**representation invariant:**

$$\begin{aligned} a \neq null \ \&\& \ b \neq null \\ a.length &= b.length \\ \forall i. 0 \leq i < b.length &\Rightarrow b[i] \geq 0 \end{aligned}$$

**abstraction function:** *The multiset contains  $b[i]$  copies of each  $a[i]$ .*

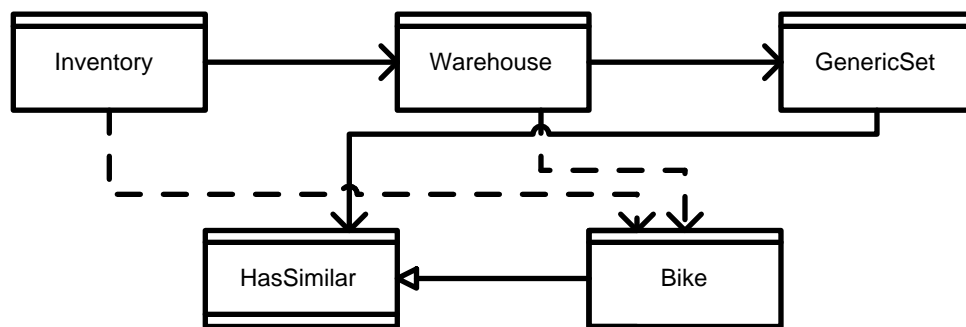
43. (6 points) Complete the object model below in order to describe alien life forms. Aliens are either male or female, and some aliens are fire-breathing. Aliens can have many heads, but each alien must have at least one head. Heads are never destroyed, nor are new ones created. After an alien dies, its head(s) detach and can at some later time become part of another alien that is being born. But once an alien is born, it cannot gain new heads or lose any heads. Aliens don't share heads. Your object model should not express any other properties about aliens.



44. (6 points) You have been hired to review the design of the database for Mike's Bikes. To analyze the quality of the design, you decide to construct a module dependence diagram for this system.

Mike has outlined the code he plans to use; see page 12, which you may tear off for easier reference. Unfortunately for you, the outline is incomplete, and with hardly any specifications. Assuming the implementation is well-designed, with no unnecessary dependences, construct an MDD by completing the below figure.

You will have to make educated guesses as to how the classes are implemented and what the methods do. In order to answer this question, you need to understand the classes at a high level, but do not need to understand every nuance of the implementation.



Warehouse *does not depend on* HasSimilar *even though it presumably calls* GenericSet.hasElement(HasSimilar), *since it calls it with a* Bike. Warehouse *only weakly depends on* Bike *because it compares Bikes with* GenericSet. GenericSet *doesn't depend on* Bike *at all because it uses* HasSimilar's *spec for equality*.

This is the code you will analyze for question 44, which is stated on page 11.

```
class Inventory { // mutable
    // implementation fields here

    public Inventory();
    public int addWarehouse(Warehouse r);
    public void addBike(int warehouseNumber, Bike b);
    public Warehouse getWarehouse(int warehouseNumber);
    public Warehouse findBike(Bike b);
}

class Warehouse { // mutable
    GenericSet bikes;

    public Warehouse();
    public void addBike(Bike b);
    public boolean hasBike(Bike b);
}

class GenericSet { // mutable
    // implemented via dynamically resized array.

    public GenericSet();
    public void addElement(HasSimilar obj);
    // returns true if this contains an element e such that e.similar(obj)
    public boolean hasElement(HasSimilar obj);
}

interface HasSimilar {
    // returns true if the 'this' and 'obj' cannot
    // be distinguished based on calls to their observers.
    public boolean similar(HasSimilar obj);
}

class Bike implements HasSimilar { // mutable
    // various fields here

    public Bike(/* various attributes here */);
    // getter and setter methods for attributes here
    public boolean similar(HasSimilar obj);
}
```