

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
6.170 LABORATORY IN SOFTWARE ENGINEERING  
FALL 2000

Quiz, Part 2  
SOLUTIONS  
November 1, 2000

There are 3 sections (labeled A through C), forming 8 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 or 0 in the appropriate box on the answer sheet.

### *Object Models and Module Dependences*

1. Every set in a problem object model must become a class in the code model. *False.*
2. Arrows in a problem object model show navigation direction. *False; problem object models say nothing about 'navigating' between objects. The significance of the order is purely semantic: in the relation Parent, it distinguishes the parent from the child, for example.*
3. A problem object model usually represents an infinite collection of configurations. *True.*
4. An arrow from *A* to *B* in the code model implies at least a weak dependence of *A* on *B*. *True.*
5. A strong dependence of *A* on *B* implies an arrow from *A* to *B* in the code model. *False; it can be due to a call on a local variable.*
6. If *A* depends on *B* and *B* depends on *C*, then *A* depends on *C*. *False.*
7. If *A* depends on *B*, then *B* depends on *A*. *False.*
8. If *A* depends on *B*, and *B* depends on *A*, there must be a recursive method call in *A* or *B*. *False.*
9. A class cannot have a strong dependence on *Object*. *False.*
10. Dependences can be inferred from the code object model alone. *False.*
11. Eliminating dependences is a design aim. *True.*
12. Introducing interfaces can help eliminate dependences. *True.*
13. In the code object model, a relation may not connect a set to itself. *False.*

### *Specifications*

14. A strong precondition can allow a more efficient implementation of a method. *True.*
15. Strong preconditions tend to simplify client code. *False.*
16. Every precondition should be checked at runtime. *False.*
17. A postcondition must specify exactly one post-state for each pre-state. *False; it can be non-deterministic.*
18. *Modifies x* means that the object *x* is always modified. *False.*
19. Specifications of *immutable* datatypes tend to have fewer *modifies* clauses. *True.*
20. The postcondition must specify at least one post-state for each pre-state satisfying the precondition. *True.*
21. A declarative specification is one that declares every argument type. *False.*
22. The client of a method is responsible for establishing the precondition. *True.*

### *Rep invariants*

23. Rep invariants should be included explicitly in the preconditions of public methods. *False; they are an implementation concern only.*
24. Rep invariants and abstraction functions are designed to make local reasoning possible. *True.*
25. Local reasoning depends on absence of rep exposure. *True.*
26. Java guarantees absence of rep exposure. *False.*
27. The Java collection types suffer from a rep exposure problem. *True; discussed in the lecture on equality.*
28. The implementation of an immutable abstract type cannot suffer from rep exposure. *False; can still have a mutable rep.*
29. A mutator must ensure the rep invariant holds in the post-state, whatever the pre-state. *False.*
30. The rep invariant should hold at every point in every method of a class that implements a datatype. *False.*
31. The rep invariant should hold at the start and end of every method of a class that implements a datatype. *False; private methods.*

#### *Abstraction Functions*

32. The specification of an abstract type should include the abstraction function. *False; can't be written until rep is known.*
33. An abstraction function must map every rep value to an abstract value. *False; only those that satisfy rep invariant.*
34. An abstraction function must not map two rep values to the same abstract value. *False.*
35. It is possible to construct a legal abstraction function that maps every rep value to two abstract values. *False; it's a function on the representation values that satisfy the invariant.*
36. For an efficient implementation, ensure the abstraction function is efficiently computable. *False.*
37. Different representations require different abstraction functions. *True.*
38. Abstraction functions are not generally of interest to users of a type. *True.*
39. A type and its subtype must have the same abstraction function. *False; subtype will have augmented rep, or different rep.*
40. A class and its subclass must have the same abstraction function. *False; subclass will have additional fields to be mapped.*

## B Subtyping

The following fragments of code or specification show two classes, *MIT* and *University*. For each case, say (a) whether *MIT* is a true subtype of *University* and (b) whether *MIT* is a Java subtype of *University* by entering 0 (no) or 1 (yes) in the appropriate box.

For brevity, some methods and fields have been omitted; you may assume that methods and fields that are missing from both classes are identical in the two classes. You should assume the existence of the following classes:

```
class Student { ... }  
  
// Geek is a true and Java subtype of Student  
class Geek extends Student { ... }
```

*X* is a Java subtype of *Y* if the Java compiler permits code that might cause objects of class *X* to be bound to variables declared to have type *Y*.

**1**  
class University {  
 // requires: s.bankBalance > 10000  
 void payTuition(Student s);  
}  
class MIT extends University {  
 // requires: s.bankBalance > 20000  
 void payTuition(Student s);  
}  
*not a true subtype because of incompatible specs, but Java allows it*

**2**  
class University {  
 // effects: s.salary' > 40000  
 void graduate(Student s);  
}  
class MIT extends University {  
 // effects: s.salary' > 40000 and  
 // s.hasTechReviewSubscription' = true  
 void graduate(Student s);  
}  
*true subtype, because can strengthen postcondition, and Java allows.*

**3**  
class University {  
 // returns: number between 0 and 4.0  
 float gpa(Student s);  
}  
class MIT extends University {  
 // returns: number between 0 and 5.0  
 float gpa(Student s);  
}  
*not true subtype, since specs incompatible, but Java allows.*

**4**  
class University {  
 Integer freshmanHousingOptions();  
}  
class MIT extends University {  
 // returns: result < Integer.MAX\_VALUE  
 int freshmanHousingOptions();  
}  
*neither, since Integer and int are incomparable.*

**5**  
class University {  
 void matriculate(Student s);  
}  
class MIT extends University {  
 void matriculate(Geek g);  
}  
*a tricky one: Java treats MIT's matriculate as a fresh method, distinguished from its inherited method by overloading. so both true.*

**6**  
class University {  
 Student studentBodyPresident();  
}  
class MIT extends University {  
 Geek studentBodyPresident();  
}  
*true subtype, by contravariance, but Java does not allow it. can't overload or extend with different return types.*

**7**  
class University {  
 int daysPerSemester() { ... }  
}  
class MIT extends University {  
 int daysPerSemester() {  
 return super.daysPerSemester();  
 }  
}  
*true and Java subtype.*

**8**

```
class University {
    void graduate(Student s);}

class MIT extends University {
    void graduate(Student s) throws
        CannotSwimCheckedException;}
```

*neither, since throwing exception changes behaviour, and Java compiler rejects this.*

**9**

```
class University {}

class MIT extends University {
    void getUROP(Geek g);}
```

*both, since behaviour preserved on existing methods.*

**10**

```
class University {
    // returns: result < 8
    int allNighterHoursSlept();}

class MIT extends University {
    // returns: 0
    int allNighterHoursSlept();}
```

*both; can strength post by resolving non-determinism.*

**11**

```
class University {
    // returns: a copy of this university
    University copy();}

class MIT extends University {
    // returns: a copy of this university
    MIT copy();}
```

*true subtype, by contravariance, but Java rejects because can't overload or extend with different return types.*

**12**

```
class University {
    // returns: number of cinemas on campus
    int movies();}

class MIT extends University {
    // returns: the number of LSC shows/year
    int movies();}
```

*not a true subtype since specs differ, but a Java subtype since difference is not observable to compiler.*

**13**

```
class University {
```

```
    // modifies: s.bankBal
    void matriculate(Student s);}
```

```
class MIT extends University {
    // modifies: s.bankBal, s.geekQuotient
    void matriculate(Student s);}
```

*not a true subtype since subclass method may do modifications not permitted by superclass method; Java allows.*

**14**

```
class University {
    // returns the department of Music
    Department musicDepartment();}

class MIT extends University {}
```

*true and Java subtype, since method is inherited.*

**15**

```
class University {
    // requires: s.gpa and s.athleticAbility
    // above threshold
    void playSport(Student s);}
```

```
class MIT extends University {
    // requires: s.gpa above threshold
    void playSport(Student s);}
```

*true and Java subtype, since subtype can weaken precondition.*

**16**

```
interface University {}

class MIT implements University {}
```

*true and Java subtype.*

**17**

```
interface NonProfit {}

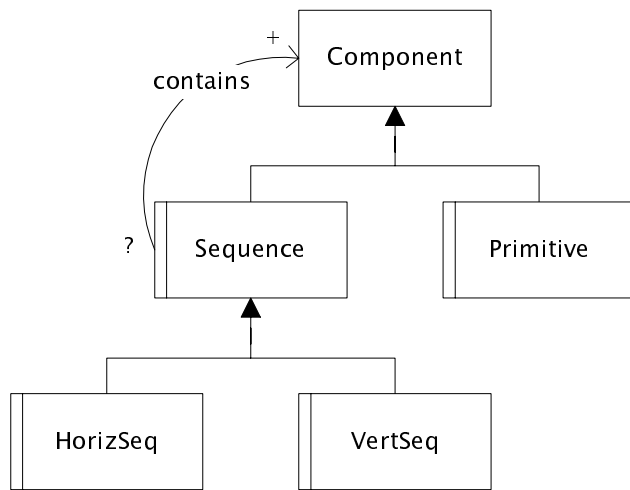
class University implements NonProfit {
    University asUniversity() { return this; }}

class MIT implements NonProfit {
    University asUniversity() {
        return (University)this;}}
```

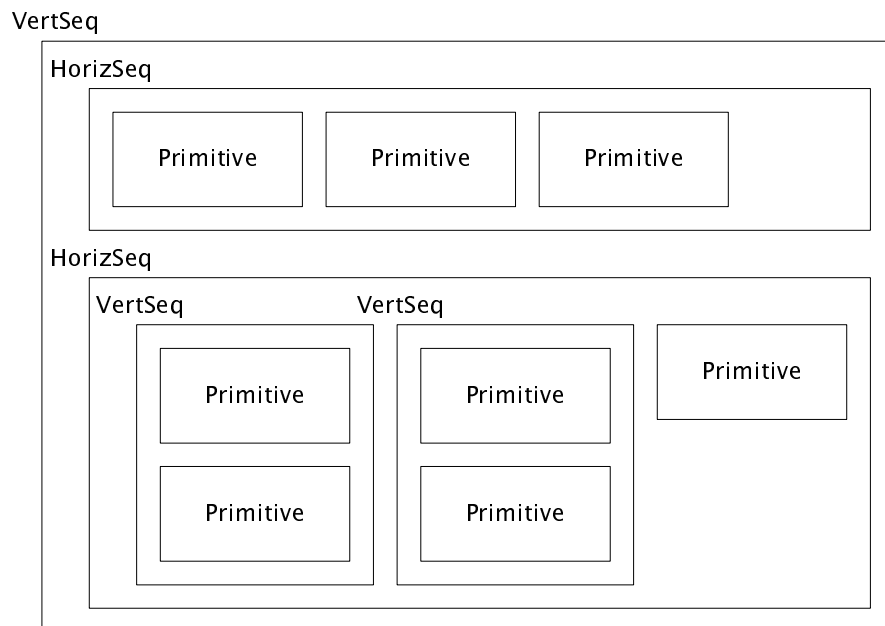
*a true subtype, since MIT's objects conform to the spec of University's objects. Java rejects though, because the cast is illegal. We decided this was too sneaky, so we didn't grade it.*

## C Design Patterns

As part of your design of the formatting subsystem of a web browser, you have constructed the following code object model:



A component to be displayed is either a horizontal sequence, a vertical sequence or a primitive component. A horizontal sequence consists of a sequence of components that will be arranged horizontally; the components of a vertical sequence will be arranged vertically. Here is a sample layout:



1 What design pattern is being used here? Write its name in the box.

*Composite.*

In your design, you intend these object model constraints to hold:

- (a) The components form a tree, with no sharing of subtrees.
- (b) The leaves of the tree are always primitive components.
- (c) Horizontal sequences contain only vertical sequences and primitive components. Vertical sequences contain only horizontal sequences and primitive components.

2 Which is already expressed in the object model diagram? Write up to three of the letters (a), (b), (c) in the box, or *none* if you think none are expressed.

*ab.*

3 How typical are these constraints of the pattern? Rank the three constraints by entering a string of three letters in the box from most to least typical. For example, if you think that (a) is representative of uses of this pattern, (b) holds occasionally, and (c) is specific to this problem, you'd write *abc*.

*bac.*

4 What dependences would you expect this pattern to exhibit? Complete the matrix by writing *S* for a strong dependence, *W* for a weak dependence and a dash for no dependence. (Note that the diagonal elements have been shaded; you should not fill them in.)

*The important insight is that the pattern eliminates dependences (as many patterns do). The only possible dependences are on Component, and perhaps of HorizSeq and VertSeq on Sequence. We graded this by taking away one point for each mark that was not a dash in any of the other matrix entries.*

5 Each kind of component has a *width* method that returns the width of the component in pixels. Where should this method be implemented?

- (a) in *Component* only
- (b) in *Sequence* and *Primitive* only
- (c) in *HorizSeq*, *VertSeq* and *Primitive*.

Enter one of (a), (b) or (c) in the box.

*In (c), since the width of vertical and horizontal components must be computed differently.*

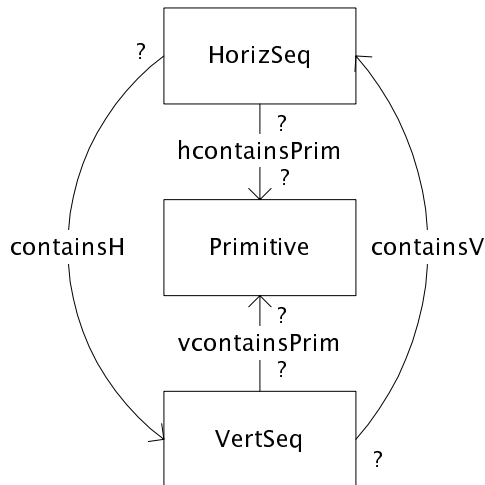
Sequence components offer an *addElement* mutator method. You are trying to decide whether it should be declared in *Component* (and perhaps implemented elsewhere). Which of the consequences might this have?

For each question 6 thru 9, enter 0 (false) or 1 (true) in the appropriate box.

- 6 Declaring *addElement* in *Component* risks making *Primitive* not a subtype of *Component*
- 7 Declaring *addElement* in *Component* may allow clients to use components more uniformly
- 8 Declaring *addElement* in *Component* may result in compile-time checks finding fewer bugs
- 9 Declaring *addElement* in *Component* forces *Component* to implement the *contains* relation

*All true except 9.*

Your colleague Fred Foolish, who is ignorant of design patterns, claims that the following structure is preferable:



He explains that each horizontal sequence object, for example, holds either a primitive component or a sequence of vertical sequence objects. This means that a horizontal sequence of several primitive components would be represented by a horizontal sequence of vertical sequences, each of which contains a single primitive component.

To explain why he's wrong, you make a number of points. Which of the points below is a sound claim about the *design pattern version*? For each question 10 thru 17, enter 0 (false) or 1 (true) in the appropriate box.

- 10 The design pattern version is likely to have a weaker representation invariant.
- 11 The design pattern version has less coupling between modules.
- 12 The design pattern version is likely to require fewer casts and uses of *instanceOf*.
- 13 The design pattern version can be extended with new kinds of component more easily.
- 14 The design pattern version admits the possibility of greater code sharing.
- 15 The design pattern version is likely to result in simpler and more uniform interfaces.
- 16 The design pattern version makes it easier to ensure that horizontal components don't contain horizontal components.

*All true except 16.*