

# Midterm

CMSC 433

Programming Language Technologies and Paradigms

Fall 2003

October 23, 2002

## Guidelines

This exam has 8 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand and I will come to you. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. Errors on the exam will be posted on the board as they are discovered. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Points	Score
1	20	
2	15	
3	25	
4	15	
5	25	
Total	100	

1. (Short Answer, 20 points)

(a) (Subtyping, 4 points) What does it mean for  $A$  to be a subtype of  $B$  (written  $A \leq B$ )?

**Answer:**

*$A \leq B$  implies that a value of type  $A$  can be used wherever a value of type  $B$  is expected.*

(b) (Subtyping, 4 points) What are the two ways in Java to induce the subtyping relationship  $A \leq B$ , where  $A$  and  $B$  are non-primitive types?

**Answer:**

*The first is inheritance: given a class  $B$ , we can define class  $A$  to extend class  $B$ . The second is interfaces: given an interface  $B$ , we can define class  $A$  to implement interface  $B$ .*

(c) (Design Patterns, 4 points) A goal of design patterns is to remove dependencies between classes. Briefly describe a general principle underlying all design patterns works to achieve this goal.

**Answer:**

*Programming to an interface, rather than an implementation: this makes it so that clients of the interface depend only on the interface, and therefore the implementation of that interface can be easily changed. A key example of this idea is in the Abstract Factory pattern.*

*It was also acceptable say favor composition over inheritance: a child depends on its parent's implementation, and this dependency can be broken by using composition instead.*

(d) (Testing, 4 points) Name one advantage and one disadvantage of automatic testing frameworks like JUnit, relative to more ad hoc approaches, like manually examining printed output.

**Answer:**

*Some advantages are (1) that tests can be re-run easily; (2) over the long term, this reduces total time testing; (3) tests are separated from a particular implementation. Disadvantages are (1) tests can be harder to write; (2) there may be less useful feedback, since you only learn that the test failed, but few details in how it failed.*

(e) (Refactoring, 4 points) You should re-run a program's test cases each time it is refactored. What about the process of changing code by refactoring makes this reasonable, as compared to adding a new feature?

**Answer:**

*Refactoring is about improving design, as opposed to changing functionality. Therefore, while the code is changed, the program should still do the same thing, meaning we can re-run all of our existing tests to make sure it actually does.*

Here we describe some code for conducting financial transactions in multiple currencies, which you will use in the next two questions. The code contains one interface (`IMoney`) and two concrete classes (`Money` and `MoneyBag`), which implement this interface.

The `IMoney` interface describes the basic currency operations (adding currencies). Each method does essentially the same thing: given an object representing some money, the method returns a new object that represents the sum of the current (`this`) object with the one passed as an argument. As such, classes that implement this interface are essentially *immutable*.

```
public interface IMoney {
    IMoney add(IMoney im);
    IMoney addMoneyBag(MoneyBag b);
    IMoney addMoney(Money m);
}

public class Money implements IMoney {
    public Money(int amount,String currency);
    public int amount();
    public String currency();
    public IMoney addMoney(Money m);
    public IMoney addMoneyBag(MoneyBag b);
    public IMoney add(IMoney m);
    public boolean equals(Object o);
}
```

The `Money` class represents a sum of money in a particular currency. For example, `Money` instances representing 50 US dollars, 20 US dollars, and 30 Swiss Francs would be created as follows:

```
Money aMoney1 = new Money(50, "USD");
Money aMoney2 = new Money(20, "USD");
Money aMoney3 = new Money(30, "CHF");
```

The methods `amount` and `currency` return the elements passed to the constructor. When two `Money` instances of the same currency are added together, the result is a new `Money` object of the same currency whose value is the sum of the two old values.

```
IMoney aMoney4 = aMoney1.add(aMoney2); // result: aMoney4 is Money(70,"USD")
```

Two `Money` instances of different currencies cannot be directly added together. Instead, they are “virtually” added by creating and returning an instance of the `MoneyBag` class containing both `Money` instances.

```
public class MoneyBag implements IMoney {
    public MoneyBag(Money m1, Money m2);
    public MoneyBag(Money m, MoneyBag bag);
    public MoneyBag(MoneyBag m1, MoneyBag m2);
    public IMoney addMoney(Money m);
    public IMoney addMoneyBag(MoneyBag s);
    public IMoney add(IMoney m);
    public boolean equals(Object o);
}
```

A `MoneyBag` is a set of `Money` instances of different currencies, with at most one instance per currency. The three constructors create a `MoneyBag` from existing `Money` or `MoneyBag` instances by adding their contents together. As mentioned above, `MoneyBag` instances can also be created by adding together two `Money` instances with different currencies:

```
IMoney aMoneyBag5 = aMoney2.add(aMoney3);
// result: aMoney4 contains Money(20, "USD") and Money(30, "CHF")
```

Adding two `MoneyBag` instances  $x$  and  $y$  results in a new `MoneyBag` instance  $z$ . Currencies that appeared in both  $x$  and  $y$  will be summed together in  $z$ . For those currencies that appeared in either  $x$  or  $y$ , but not both,  $z$  will also have them, unchanged. For example, if  $x$  contains `Money(20, "USD")` and `Money(30, "CHF")` and  $y$  contains `Money(40, "USD")` and `Money(20, "DM")`, then  $z$  will contain `Money(60, "USD")`, `Money(30, "CHF")`, and `Money(20, "DM")`.

2. (Testing, 15 points)

When an executable statement is executed during a test run, we say that the statement was covered. The number of covered statements in the program divided by the total number of executable statements is called the coverage level.

For each public method in the following implementation of the `Money` class, you wish to achieve 100% statement coverage (that is, every statement in the method is executed at least once). Therefore, a hypothetical test suite would have to call each method one or more times, using different parameters for each call as necessary. Indicate for each method the invocations you would need for 100% coverage. Show your results in a 2-column table: one for the method, and one for each description of an invocation's parameters' properties.

```
public class Money implements IMoney {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount = amount; fCurrency = currency;
    }

    public int amount() {return fAmount;}
    public String currency() {return fCurrency;}

    public IMoney addMoney(Money m) {
        if (m.currency().equals(currency()) )
            return new Money(amount()+m.amount(), currency());
        return new MoneyBag(this, m);
    }
    public IMoney addMoneyBag(MoneyBag s) {return s.addMoney(this);}

    public IMoney add(IMoney m) {return m.addMoney(this);}

    public boolean equals(Object anObject) {
        if (anObject instanceof Money) {
            Money aMoney = (Money)anObject;
            if (aMoney.currency().equals(currency())) {
                if (amount == aMoney.amount())
                    return true;
            }
        }
        return false;
    }
}
```

Answer:

<i>Method</i>	<i>Inputs</i>
<i>amount()</i>	<i>none</i>
<i>currency()</i>	<i>none</i>
<i>addMoney(o)</i>	<i>o a Money object with the same currency type</i> <i>o a Money object with different currency type</i>
<i>addMoneyBag(o)</i>	<i>o is any MoneyBag object</i>
<i>add(o)</i>	<i>o is any IMoney object</i>
<i>equals(o)</i>	<i>o is a non-Money object</i> <i>o is a Money object with the same amount and same currency as this</i>

3. (Testing, 25 points)

Write a Junit test class called MoneyTest. MoneyTest should include the following test methods:

testMoneyEquals	Test that Money.equals() properly deems equality and inequality with IMoney instances.
testSimpleAdd	Test that two Money instances of the same currency are added correctly.
testBagEquals	Test that MoneyBag.equals() properly deems equality and inequality with IMoney instances.
testMixedSimpleAdd	Test that a Money instance and a MoneyBag instance are added correctly, both for the case that they each have the same currency, and for the case that they have different currencies.

**Answer:**

```
import junit.framework.*;

public class MoneyTest extends TestCase {
    private Money f12CHF = new Money(12, "CHF");
    private Money f14CHF = new Money(14, "CHF");
    private Money f7USD = new Money( 7, "USD");
    private Money f21USD = new Money(21, "USD");
    private MoneyBag fMB1 = new MoneyBag(f12CHF, f7USD);
    private MoneyBag fMB2 = new MoneyBag(f14CHF, f21USD);
    private MoneyBag fMB3 = new MoneyBag(new Money(12,"DM"),fMB2);

    public void testMoneyEquals() {
        assertEquals(f12CHF, new Money(12, "CHF"));
        assertTrue(!f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(fMB1));
    }

    public void testSimpleAdd() {
        assertEquals(new Money(26, "CHF"),f12CHF.add(f14CHF));
    }

    public void testBagEquals() {
        assertEquals(fMB1, new MoneyBag(f12CHF, f7USD));
        assertTrue(!fMB1.equals(f12CHF));
        assertTrue(!fMB1.equals(fMB2));
    }

    public void testMixedSimpleAdd() {
        assertEquals(new MoneyBag(new Money(26,"CHF"),f7USD),fMB1.add(f14CHF));
        assertEquals(fMB3,fMB2.add(new Money(12,"DM")));
        assertEquals(new MoneyBag(new Money(26,"CHF"),f7USD),f14CHF.add(fMB1));
        assertEquals(fMB3,new Money(12,"DM").add(fMB2));
    }
}
```

4. (Design Patterns, 15 points)

Configuration management systems (CMS) store snapshots of software projects so that the state of the software at a given time can be recreated at a later time—e.g., let’s say someone makes a bunch of bogus changes to the software and nothing works anymore. With a CMS system you could just delete the current software and get an earlier working copy from the repository.

Suppose that we have the following API for checking code into the CMS:

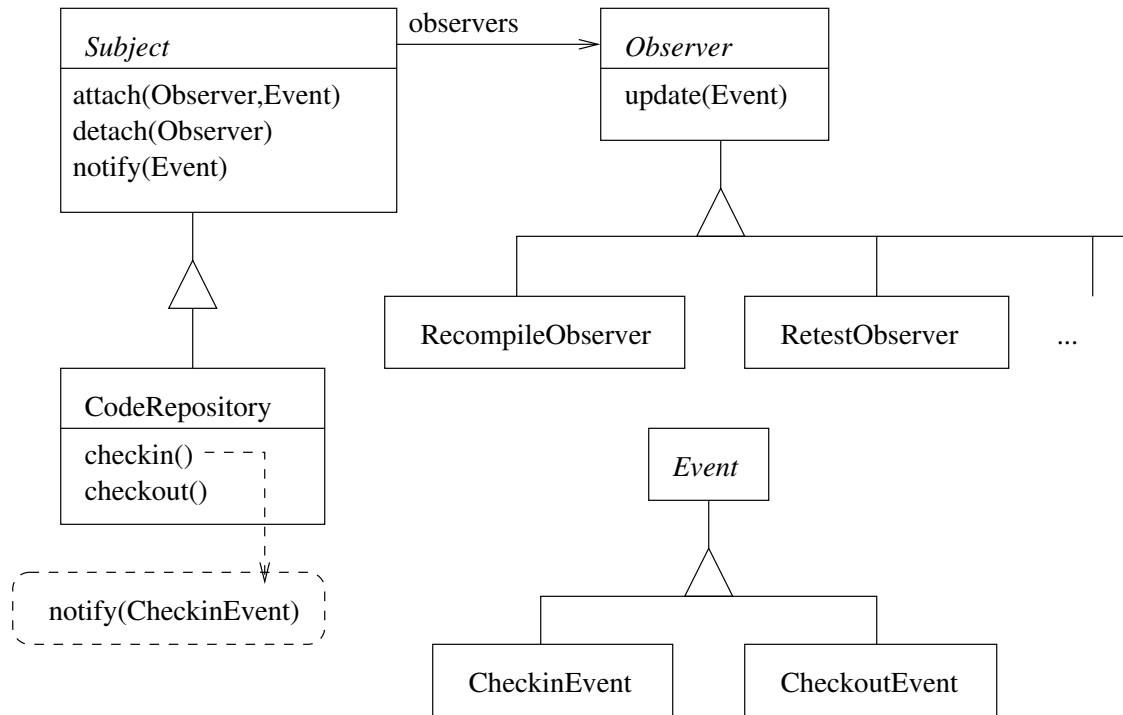
```
public class CodeRepository {
    void checkin (CodeUnit c, Project p);
    CodeUnit[] checkout (Project p);
}
```

You are a software developer who team uses the `CodeRepository` class. You realize that whenever you or your team members check in or check out code, you manually invoke several development tools. For example, you run a formatter, execute Junit tests and collect software size measures on check-in and you recompile the software on check-out.

You would like to create an automated method for invoking tools at the appropriate times. You can use the observer pattern. In particular, the `CodeRepository` object is the *subject*, and classes that perform various operations on check-in or check-out will be the *observers*. Observers may be interested in either check-ins, check-outs, or both.

In the space below, present a class diagram that sketches your solution. Include pseudocode as necessary to illustrate the actions of the classes.

**Answer:**



*An alternative approach would be to skip the `Event` class and instead have the `Observers` define `checkin` and `checkout` methods, and then have two separate notification routines in the abstract subject, one for each event.*

5. (Design Patterns, 25 points)

Below we define a class hierarchy for simple expressions: an interface `Expression`, and three classes that implement it: `Constant` for constants, `AddExpression` for additions of two expressions, and `IncrementExpression` for adding one to (i.e. incrementing) an expression. Write a visitor class called `DeincrementVisitor` that, when told to visit some expression  $e$ , creates a new expression  $e'$  such that all `IncrementExpression` instances that occur in  $e$  are replaced by equivalent `AdditionExpressions`. You can write an internal or external visitor, as you choose. You must fill in the `accept` methods below to reflect your choice.

**Answer:**

*The code below performs the traversal within the visitor, and so used the visitor control stack to store temporary results. Doing the traversal in the `Expression` subclass `accept` methods would require the use of a stack in the visitor. You could also use an explicit stack even when the visitor does the traversal.*

```
public interface Expression {
    void accept(Visitor v);
}

public class Constant implements Expression {
    int value;
    public Constant(int v) { this.value = v; }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class AddExpression implements Expression {
    Expression l, r;
    public AddExpression(Expression l, Expression r) {
        this.l = l; this.r = r;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class IncrementExpression implements Expression {
    Expression e;
    public IncrementExpression(Expression e) {
        this.e = e;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public interface Visitor {
    void visit(Constant c);
    void visit(AddExpression e);
    void visit(IncrementExpression e);
}
```

*The answer continues on the next page.*

```

public class DeincrementVisitor implements Visitor {
    private Expression modifiedExpression;

    public void visit(Constant c) {
        modifiedExpression = c;
    }
    public void visit(AddExpression e) {
        Expression lexp, rexp;
        e.l.accept(this);
        lexp = modifiedExpression;
        e.r.accept(this);
        rexp = modifiedExpression;
        modifiedExpression = new AddExpression(lexp, rexp);
    }
    public void visit(IncrementExpression e) {
        Expression exp;
        e.e.accept(this);
        exp = modifiedExpression;
        modifiedExpression = new AddExpression(new Constant(1), exp);
    }
    public Expression getExpression() {
        return modifiedExpression;
    }
}

```