Midterm

CMSC 433
Programming Language Technologies and Paradigms
Fall 2004
October 28, 2004

Guidelines

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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
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) (Data Abstraction, 4 points) What is an abstraction function? What are representation invariants?

**Answer:**

An abstraction function is a function that specifies how the representation of an abstract data type is interpreted as an abstract value. Representation invariants are constraints that characterize whether a concrete instance of an abstract data type is well formed.

(c) (Design Patterns, 4 points) One principle underlying design patterns is to favor composition over inheritance. Suppose you have a class B whose functionality you want to extend. You do this by creating a new class A that contains and uses instances of B. Name 2 advantages this has over just making A a subclass of B.

**Answer:**

A depends on the interface of B, but not its implementation. So B can change its implementation without affecting users of A. The relationship between A and B is non-static and can be changed at runtime.

(d) (Testing, 4 points) Describe one advantage white-box testing has over black-box testing. Describe one advantage black-box testing has over white-box testing.

**Answer:**

White-box testing can detect problems handling special cases (or along specific paths). Black-box testing can reveal missing path problems (i.e., cases the developer forgot to implement). Black-box testing scales up to large programs.

(e) (Program Analysis, 4 points) In the lectures on testing and the findBugs tool we mentioned something called a control flow graph (CFG). What do the nodes and arcs of a CFG represent?

**Answer:**

A program’s CFG has one node for every basic block (linear sequence of instructions without control jumps) and one arc between any two basic blocks that can execute in sequence.
2. (Design Patterns (coding). 20 points). Assume that your web browser services web requests using objects that implement the Request interface.

```java
public interface Request {
    String get (String arg ); /* arg is the URL */
}
```

Write a proxy class for Request, called CachedRequest, that follows the proxy design pattern. This class will maintain a cache of recently requested web pages in a HashMap indexed by their URL. When CachedRequest get() method is called, it should look for the requested page in the cache and return it if available. The constructor for CachedRequest will take the Request instance it will proxy as its argument.

Note: Use HashMap’s put (Object key, Object value) method and get (Object key) method to interact with the cache. Remember that get returns null if the key is not in the HashMap.

**Answer:**

```java
import java.util.*;

public class CachedRequest implements Request {

    private Request _request;
    private HashMap _cache = new HashMap();

    CachedRequest(Request request) {
        _request = request;
    }

    public String get(String arg) {
        String result = (String) _cache.get(arg);
        if (result == null) {
            result = _request.get(arg);
            _cache.put(arg, result);
        }
        return result;
    }
}
```
3. (Testing, 20 points). Write a set of Junit tests that achieve 100% branch coverage of the following class.

```java
public class Account {
    public long balance;

    /**
     * Withdraws exactly the passed amount from the <code>Account</code>.
     * @param amount
     * @return amount withdrawn from the <code>Account</code>
     * @throws InsufficientFundsException
     *     if the <code>Account</code> contains insufficient funds for
     *     the requested withdrawal
     * @throws IllegalArgumentException
     *     if the passed <code>amount</code> to withdraw is less than
     *     or equal to zero.
     */
    public long withdraw(long amount) throws InsufficientFundsException {
        if (amount <= 0) { throw new IllegalArgumentException(); }
        if (amount > balance) {
            throw new InsufficientFundsException(amount - balance);
        }
        balance -= amount; return amount;
    }

    /**
     * Deposits exactly the passed amount into the <code>Account</code>.
     * @param amount
     * @return amount to deposit
     * @throws ArithmeticException
     *     if requested deposit would cause the balance of this
     *     <code>Account</code> to exceed Long.MAX_VALUE.
     * @throws IllegalArgumentException
     *     if requested deposit is less than or equal to zero.
     */
    public void deposit(long amount) {
        if (amount <= 0) { throw new IllegalArgumentException(); }
        long newBal = balance + amount;
        if (newBal < 0) { throw new ArithmeticException(); }
        balance = newBal;
    }

    /**
     * Gets the current balance of this <code>Account</code>.
     * @return the current balance
     */
    public long getBalance() { return balance; }
}
```

Write your answer on the following page.
import junit.framework.TestCase;

public class AccountTestCase extends TestCase {
    Account account;

    protected void setUp() {
        account = new Account();
    }

    public void testDeposit1() {
        try {
            account.deposit(-1);
            fail("Account.deposit() didn’t throw IllegalArgumentException when negative value passed");
        } catch (IllegalArgumentException e) {} 
    }

    public void testDeposit2() {
        try {
            account.deposit(1);
            account.deposit(Long.MAX_VALUE);
            fail("Account.deposit() didn’t throw ArithmeticException when balance overflowed");
        } catch (ArithmeticException e) {} 
    }

    public void testDeposit3() {
        account.deposit(20);
        assertTrue(
                "Account.deposit() didn’t deposit 20 correctly. Resulting balance should have been 20, but " + account.getBalance() + ".",
                account.getBalance() == 20);
    }

    public void testWithdraw1() {
        account.deposit(20);
        try {
            account.withdraw(25);
            fail("Account.withdraw() should have thrown Insufficient Funds Exception");
        } catch (Insufficient FundsException e) {} 
    }

    public void testWithdraw2() {
        try {
            account.withdraw(-1);
            fail("Account.withdraw() should have thrown IllegalArgumentException");
        } catch (IllegalArgumentException e) {
            } catch (Insufficient FundsException e) {
                fail("Account.withdraw() threw Insufficient Funds Exception when it should have thrown IllegalArgumentException");
            } 
    }

    public void testWithdraw3() {
        account.deposit(20);
        try {
            account.withdraw(5);
        } catch (Insufficient FundsException e) {
            fail("Account.withdraw() shouldn’t have thrown Insufficient Funds Exception");
        }
    }
}
assertTrue(
    "Account.withdrawal() didn't withdraw 5 correctly. Resulting balance should have been 15, but
    + account.getBalance() + ".",account.getBalance() == 15);
}
4. (Design Patterns 15 points)

In class we discussed the Bridge pattern. This pattern is used when one part of your application is dependent on platform-specific implementations that are likely to change and the rest is dependent on application-specific details that are also likely to change.

Consider the following example: Your company has built a new futuristic “smart kitchen”, run by Hazel the cooking robot. Your job is to develop some of the code that Hazel will use to cook food using the smart kitchen appliances.

When you sit down to design this system, you see that many things are likely to change. For example, Hazel will need to prepare different foods, for example: fish or vegetables. New kinds of food will be added over time.

Once food is prepared, Hazel will cook it in different smart kitchen appliances using different cooking methods. For example, some foods will need to be baked in a conventional oven, some baked in a microwave oven, some broiled in a conventional oven, etc. In some cases any of several methods and appliances might work and the choice will depend on what else is cooking at that time. The set of methods and appliances will change over time as well.

Draw UML diagrams showing how you would use the bridge pattern to allow both the foods to be prepared and the appliances and methods used for cooking to vary independently.

Make sure your diagrams show the class hierarchies involved, the interfaces to each class, pseudocode showing how your classes work together (where appropriate). For now your diagrams should account for three kinds of foods: fish, steak and vegetables and three appliances with different cooking methods: oven (bake, broil and grill), grill (bake and grill) and microwave (bake).

Answer:
5. (Design Patterns, 25 points)

Below we define a class hierarchy for logging simple financial transactions: an interface `Transactions`, and three classes that implement it: `Deposit` for adding money, `Withdrawal` for removing money and `Journal` for marking the end of a business day. Write a visitor class called `NSFVisitor`. `NSFVisitor` implements the Visitor interface which is shown below. `NSFVisitor`'s `visit()` methods should identify the first Journal object in which the account balance at the end of the day is negative (if one exists). Its `getResult()` method should print out the results from the last call to `visit()`.

Assume that the account balance is initially zero when logging first starts. **Note:** Withdrawals and Deposits are stored in the log in reverse chronological order. I.e. `._prev` refers to all previous transactions. Journals are stored in chronological order. I.e. a `Journal` stores today's transactions in `._today` and refers to future Journals via `._next`.

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

public class Withdrawal implements Transactions {
    int _value; Transactions _prev;

    public Withdrawal(int v, Transactions prev) {_value = v; _prev = prev;}
    public Withdrawal(int v) {_value = v; _prev = null;} // First trans. of the day.

    public void accept(Visitor v) {v.visit(this);} // end of Withdrawal
}

public class Deposit implements Transactions {
    int _value; Transactions _prev;

    public Deposit(int v, Transactions prev) {_value = v; _prev = prev;}
    public Deposit(int v) {_value = v; _prev = null;} // First trans. of the day.

    public void accept(Visitor v) {v.visit(this);} // end of Deposit
}

public class Journal implements Transactions {
    Transactions _today, _next;

    public Journal(Transactions today, Transactions next) { _today = today; _next = next;}
    public Journal(Transactions today) { _today = today; _next = null;} // Last day in the log.

    public void accept(Visitor v) {v.visit(this);} // end of Journal
}

public interface Visitor {
    void visit(Deposit d);
    void visit(Withdrawal w);
    void visit(Journal j);
    String getResult();
} // end of Visitor

Write your implementation of the `NSFVisitor` on the next page.
public class NSFVisitor implements Visitor {

    public void visit(Deposit d) {

    }

    public void visit(Withdrawal w) {

    }

    public void visit(Journal j) {

    }

    public String getResult () {

    }
}
Answer:

```java
public class NSFVisitor implements Visitor {

    int runningBalance = 0;
    Transactions violator = null;

    public void visit(Deposit d) {
        if (d._prev != null) {
            d._prev.accept(this);
        }
        runningBalance += d._value;
    }

    public void visit(Withdrawal w) {
        if (w._prev != null) {
            w._prev.accept(this);
        }
        runningBalance -= w._value;
    }

    public void visit(Journal j) {
        j._today.accept(this);
        if (runningBalance < 0) {
            violator = j;
        } else {
            if (j._next != null) {
                j._next.accept(this);
            }
        }
    }

    public String getResult () {
        if (runningBalance < 0 ) {
            return "Violator is " + violator;
        } else {
            return "No violations found";
        }
    }
}
```