

Exam 1

CMSC 433

Programming Language Technologies and Paradigms
Fall 2002

October 17, 2002

Guidelines

Put your name and class account number 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.

1. (8 points) In class we discussed two principles of design patterns: (1) Program to an interface, not to an implementation, and (2) favor composition over inheritance. Explain each of these principles and their advantages in your own words. Use no more than 50 words per principle.

- (a) Interfaces are based on the operations supported by a class. Class definitions are based on the implementations. Programming to the interface insulates users from implementation differences, programming to implementations does not. The former approach improves extensibility and reuse.
- (b) class inheritance and object composition are two approaches to reuse. Inheritance involves reusing a parent class' implementation. So when parent class changes, child class will be affected. This limits extensibility. Object composition reuses class functionality (programming to its interface) which should allow greater extensibility.

2. (8 points total) The design patterns described in class often involve both abstract classes and concrete classes.

- (a) (2 points) Technically speaking, how does an abstract class differ from a concrete class? In particular, what can you do with a concrete class that you can't do with an abstract one?

You can create an instance of a concrete class, but not an abstract class.

- (b) (4 points) Name the two mechanisms in Java that you can use to implement the concept of abstract classes. Name one benefit of using one mechanism over the other.

- interface: gives the protocol for a class. Individual classes can implement multiple interfaces.
- abstract class: defines a class that can be instantiated. Allows methods and variables to be defined.

- (c) (2 points) Name two examples of conceptual abstract classes (using either of Java's two mechanisms), taken either from the projects or from the Java class libraries.

Log and HttpHandler

3. (17 points) In Project 2, you wrote a web server whose behavior was implemented by subclasses of the class `HttpHandler`; for your reference, we have provided `HttpHandler`'s code below. This class has a method called `handleRequest` that, along with other methods, implements the basic algorithm for handling a web request. At the moment, handling a request consists of two parts: matching the request, and if the match occurs, writing a response in HTML.

Describe how you would modify the project code (which classes and how) so that handling a request occurs in *three* phases: matching the request, generating output in a generic format, and then formatting that output to be HTML. Use pseudocode, diagrams, text, etc. as necessary to explain your approach.

```
public abstract class HttpHandler implements Comparable {

    protected HandlerOrder orderConstraint;

    public int compareTo(Object that) {
        if (that instanceof HttpHandler) {
            HttpHandler handler = (HttpHandler)that;
            return (this.orderConstraint.compareTo(handler.orderConstraint));
        }
        throw new ClassCastException();
    }

    public boolean handleRequest(HttpServletRequest request,
        StringBuffer out) throws HandlerException {
        if (matchRequest(request)) {
            writeResponse(out);
            return true;
        }
        return false;
    }

    public abstract boolean isExclusive();

    protected abstract boolean matchRequest(HttpServletRequest request);

    protected abstract void writeResponse(StringBuffer out)
        throws HandlerException;
}
```

There are various possibilities:

`HttpHandler.handleRequest()` uses the Template Method. You can split `writeResponse()` into two parts, `processRequest()` and `writeHTMLOutput()`. We need to define a generic content container and pass an instance of it from `processRequest()` to `writeHTMLOutput()`. This function (`writeHTMLOutput()`), will have its concrete implementation in the `HttpHandler` class, to be shared by all subclasses.

Another possibility is to change the semantics of `HttpHandler` so that `writeResponse` writes a generic output. The `HttpResponse` class then formats this generic output into HTML.

A key in all solutions is that the changes must be made to the `HttpHandler` class; just saying “we could implement the subclasses of `HttpHandler` differently” is not the whole way there, since we are not enforcing the behavior we want in the superclass. Another key point is that generating HTML from the generic format should only be implemented once, not once for each subclass; this is why the output is considered “generic.”

4. (17 points) Referring once again to Project 2, recall that all handlers return their output in HTML. Suppose that you want to be able to write web servers that not only output HTML, but output other formats as well, e.g. plain text, audio, etc. Consider the effect of this change on the web server architecture: in the original server, only the function of the handlers (e.g. `GetRecordsHandler`, `LogEventHandler`, etc.) varies; in the proposed situation, both the function of the handlers, and the way handler output is formatted (e.g. as HTML, plain text, postscript, etc.) can vary.

Assuming that the way its output is formatted is determined at the time a handler is created (and not by the `BasicServer`), how would you implement this change so that new handlers and new formatting algorithms can be added with a minimum amount of code? What design pattern(s) could you use? Use class diagrams, pseudocode, text, etc. to illustrate your answer. Justify your design choices.

Various possibilities. The key is that a change to a formatting method should not result in having to reimplement each handler class, and vice versa. This would imply having a hierarchy of formatting classes separate from the hierarchy of handlers, so that they could evolve independently. You were not allowed to modify the `BasicServer`; output must be determined by the handler's constructor.

One possibility is to use the strategy pattern. Create a class hierarchy that encapsulates the formatting approach (with abstract superclass `Formatter`). When creating a handler pass it an instance of a concrete subclass of `Formatter`. Store this in an instance variable called `formatter` (for example). Split `handleRequest` into two parts as mentioned in the previous question. `WriteOutput()` will now delegate its formatting operation to `this.formatter`, rather than fixing on HTML. You could also think of this as the Bridge pattern.

A more specific solution be to implement this Strategy via the Visitor pattern. In particular, if allow our generic output to be "visitor-enabled", then `Formatter` can simply be a kind of `Visitor`, with each subclass visiting the generic output and formatting it differently.

5. (25 points) Write a Junit test class that tests implementations of the `Log` interface from Project One.¹ Your class should have three different test cases, with one no-argument method for each. Include comments with each test case indicating what it intends to test. Your test class should include all the necessary code needed to be compiled correctly. You do not need to add a `main` method.

The `Log` interface:

```
public interface Log {
    public void add(LogRecord record);
    public LogRecord[] getAll(long windowMS);
    public void setFilter(String pattern);
}
```

If you thought of useful tests, we tried to give you most of the points. We tried not to be too concerned with junit syntax, etc. In short, each test needed to test `Log`, and not `LogRecord`, or some other unrelated feature (like whether `a == b`). Each test needed to be at least somewhat different from the other tests. The test had to actually work; that is, you needed to test behavior that the `Log` could implement. For example, a number of people tried to add records to the log and then call `setFilter` (which would have no effect on records already in the log); these tests lost points.

You could also lose small amounts of points for the following mistakes. You also needed to not assume that the tests would be run in any given order; therefore you probably needed to reset a shared log between tests, or else not assume that the log had a particular set of contents. Also your tests should not have assumed that records returned by `getAll` would be returned in the order they were added, or in sorted order.

There's an example on the next page.

¹The semantics of the `getAll` function changed in Project Two with regard to an input of 0; you can use either semantics in your test cases.

```

LocalLog lltmp1;
String event1, event2, event3;
LogRecord tmp1, tmp2, tmp3;

protected void setUp()
{
    event1 = "event string1";
    event2 = "event string2";
    event3 = "event string3";
    tmp1 = new LogRecord(event1);
    tmp2 = new LogRecord(event2);
    tmp3 = new LogRecord(event3);
}

// tests that an old record is discarded from a full log
public void testLocalLog1()
{
    lltmp1 = new LocalLog(2);
    lltmp1.add(tmp1);
    lltmp1.add(tmp2);
    lltmp1.add(tmp3);

    // assumes getAll(0) returns all records
    assertTrue(lltmp1.getAll(0).length == 2);
}

// tests that the last two records added are retained
public void testLocalLog2()
{
    lltmp1 = new LocalLog(2);
    lltmp1.add(tmp1);
    lltmp1.add(tmp2);
    lltmp1.add(tmp3);

    LogRecord[] recs = lltmp1.getAll(0);
    // make sure the results are one of the last two records
    for (int i = 0; i<recs.length; i++) {
        String event = recs[i].toString();
        assertTrue(event.equals(tmp2.toString()) ||
            event.equals(tmp3.toString()));
    }
}

// tests that the filter works
public void testLocalLog3()
{
    lltmp1 = new LocalLog(2);
    lltmp1.setFilter("event string1");
    lltmp1.add(tmp1);
    lltmp1.add(tmp2);
    lltmp1.add(tmp3);

    String hold = tmp1.toString() + "\n";

    assertTrue(hold.equals(lltmp1.toString()));
}

```

6. (25 points) Write an implementation of the Log interface (shown in Problem 5) that acts as a *security proxy* for another Log, using the Proxy pattern. The constructor for the proxy will indicate which of the underlying Log's methods will be disabled by the proxy. For example, the BasicServer of Project Two might allow a LogEventHandler to add records to its log, but not to look at that log's contents or set its filter.

The key ideas were that your class needed to implement Log, and it needed to “wrap” an underlying log. In addition, the constructor should indicate which methods would be blocked, and all possibilities should be available. Finally, for generality, the underlying Log should be passed into the constructor, rather than created by the proxy (else, you'd have to pick LocalLog or RemoteLogClient).

```
public class ProxyLog implements Log {
    private Log log;
    private boolean acceptsAdd;
    private boolean acceptsGetAll;
    private boolean acceptsSetFilter;

    public ProxyLog(Log log, boolean acceptsAdd, boolean acceptsGetAll,
                   boolean acceptsSetFilter) {
        this.log = log;
        this.acceptsAdd = acceptsAdd;
        this.acceptsGetAll = acceptsGetAll;
        this.acceptsSetFilter = acceptsSetFilter;
    }

    public void add(LogRecord l) {
        if (acceptsAdd) log.add(l);
    }

    public LogRecord[] getAll(long windowMS) {
        if (acceptsGetAll) return log.getAll(windowMS);
        else return new LogRecord[0];
    }

    public void setFilter(String pattern) {
        if (acceptsSetFilter) log.setFilter(pattern);
    }
}
```