- Testing

  (Learning goal: Given a testing issue, use a mock object to help address it.)

  You have a class `Location` with methods that include basic getters and setters of latitude and longitude values. The class `Track` maintains a list of `Location` objects, and has a corresponding `add` method. You need to test that `add` does not call any basic setter method of a `Location` object. (If such a call happens, the test should fail.) State any further assumptions.

```
class Location {
    private String latitude;
    private String longitude;

    public Location( String latitude, String longitude ) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    public void setLatitude( String latitude ) {
        this.latitude = latitude;
    }
    public void setLongitude( String longitude ) {
        this.longitude = longitude;
    }
    // getters and other methods
    …
}

class Track {
    …
    public void add( Location location ) {
        …
    }
}
```

(a) [1] Define a mock location as correct Java code. Do not change the existing `Location` or `Track` class definitions.

(b) [1] Complete the test method as correct Java code.

```
class TestTrackAdd extends TestCase {
    public void testNoLocationSetter() {
        Track track = new Track();









    }
}
```

- Design patterns

  (Learning goal: for a given design, draw a correct UML sequence diagram to describe its expected behavior.)

[3] Consider the model-view or observer design pattern, using the Java `Observable` class and `Observer` interface, combined with the command design pattern.

A Client class will create a new Command object and bind it to a Invoker object via dependency injection. The Invoker object will call upon the Command object to execute, which calls upon a model object to modify.

Draw the UML sequence diagram to describe the complete behavior of this combined pattern, including the initial command creation and eventual updated views. Include the relevant other objects, suitable messages, and method activations. State any further assumptions.

- Design patterns

  (Learning goal: for a given design problem, apply an appropriate design pattern; draw a correct UML class diagram to describe its structure.)

[4] You are modeling an integrated fire protection system for a building with floors, each floor with areas. Each kind of unit (floor or area) has an appropriate fire alert and fire suppression behavior to be triggered consistently. Describe clearly how to represent this configuration using a suitable design pattern.

Apply the pattern, and outline the structure of the design using a correct UML class diagram.

- Design patterns

  (Learning goal: given an issue in implementing or applying a design pattern, describe correctly how to address it.)

The observer design pattern is used to define a dependency between objects so that when one subject object changes state, all its dependent observer objects are notified and updated automatically.

(a) [2] How would you modify an implementation of this pattern to handle the case where an observer object may need to be notified of changes in many subject objects (not just one)?

(b) [2] How would you adjust an implementation of the observer design pattern to allow a subject object, which may have a number of observers, to be deleted? Explain clearly.

- Design patterns

(Learning goal: given a design problem, identify and justify the most suitable design pattern to help solve it.)

In the following situations, explain which design pattern is most appropriate for addressing the problem.

(a) [2] You want to develop an application to count the total size of a file-system directory. Directory sizes are the sum of the sizes of their contents.

(b) [2] You are developing a spreadsheet application that allows cells to be calculated automatically based on formulas depending on other cells.

(c) [2] You want to develop a kids' calculator for integers and you have a college math calculator.

(d) [2] You want to develop a file reader that is capable of reading a file, which can possibly be (1) zipped, (2) encrypted (3) zipped and encrypted or (4) encrypted then zipped and encrypted again.

- Refactoring

(Learning goal: given an implementation, identify the code smells and outline refactorings to address them; draw the UML class diagram for the correctly refactored design.)

Suppose there is an `Employee` class and a `PayType` class. Different employees may be paid differently. For example, a salesperson may get a commission beyond their usual monthly salary. A manager may get a management bonus.

Assume an `Employee` object has a `PayType` object that is responsible for such types of pay. Consider the following (partial) implementation of a `PayType` class.

```
class PayType {
    …
    int payAmount( Employee emp ) {
        switch (this.getTypeCode()) {
        case ENGINEER:
            return emp.getMonthlySalary();
        case SALESPERSON:
            return emp.getMonthlySalary()
                + emp.getCommission();
        case MANAGER:
            return emp.getMonthlySalary()
                + emp.getBonus();
        …
        }
    }
}
```

(a) [1] Identify the design principle violations or code smell(s) in this design.

(b) [2] Outline how to refactor this code to use polymorphism.

(c) [3] Draw the UML class diagram after the refactoring. State any further assumptions.

- Performance

  (Learning goal: given an implementation, propose a version with improved performance within specific constraints.)

[3] Consider the following method to compute the base-two logarithm of a byte value (assumed positive here, maximum 127), truncated down to the nearest integer. For example, if the byte value is 17, the method returns 4.

```
public static int logb2( byte b ) {
    return (int)( Math.log( b ) / Math.log( 2 ) );
}
```

Assume this method is heavily used, and that using operations like `Math.log( double )` is relatively slow and must be avoided.

Optimize the `logb2` method to reduce the computation time needed, while maintaining the same expected behavior and using comparable data space. Your method must be proper Java and be a self-contained implementation (not call any other methods). Add comments as appropriate to explain how it is intended to work.

Explain the worst case input for your approach.