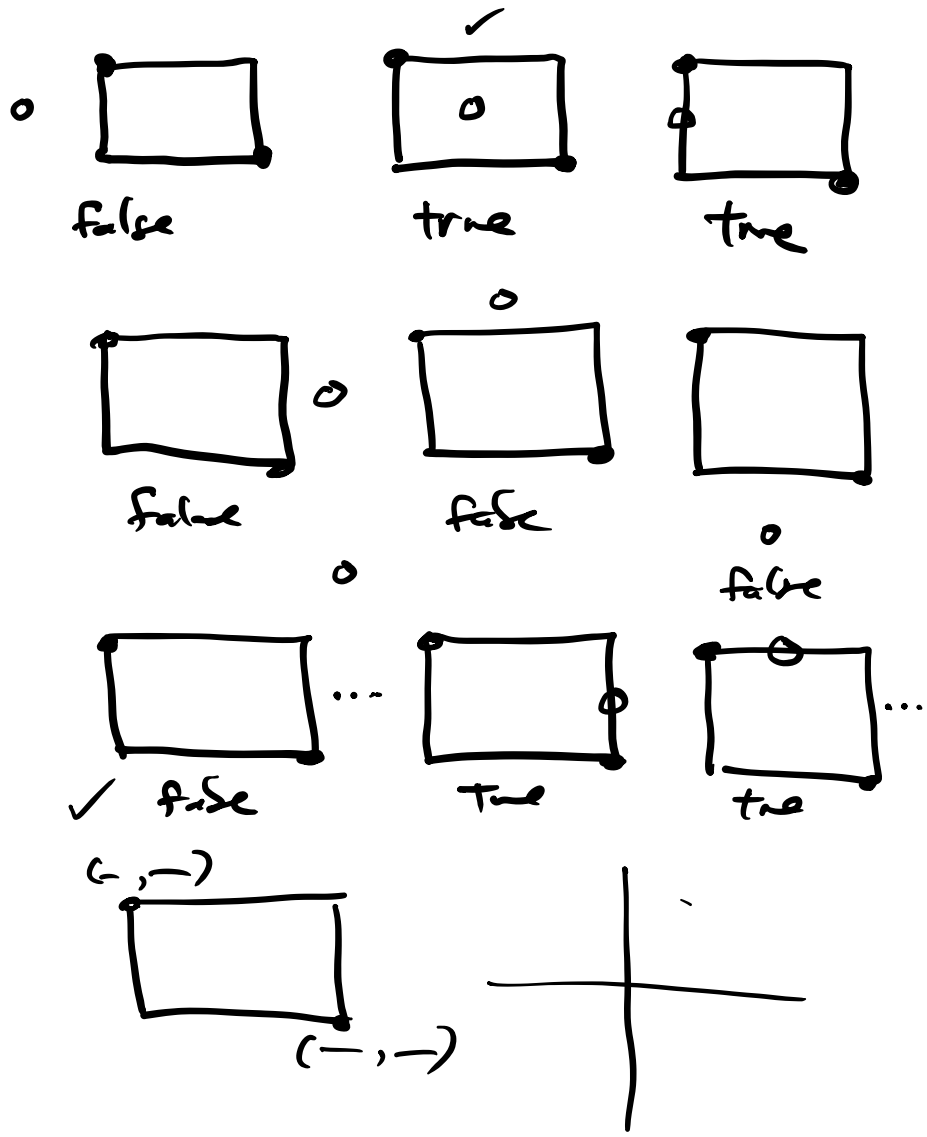
- Testing

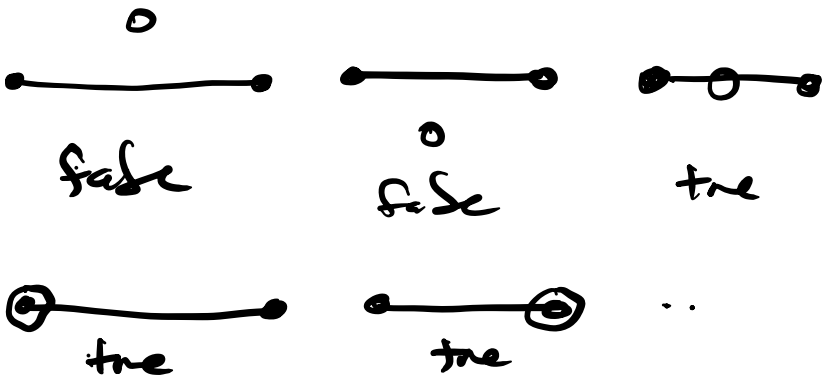
  (Learning goal: given a programmer interface, determine equivalence classes of tests to verify its required behavior.)

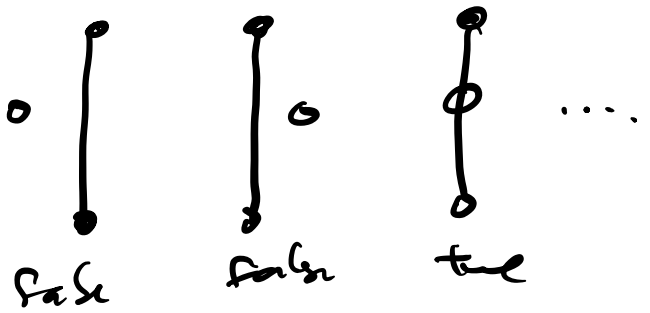
[3] Consider a `Rect` class to represent a rectangle in a two-dimensional integer plane.

```
public class Rect {
    …

    // create rectangle with given corners
    public Rect( Point topLeft, Point bottomRight ) { … }

    // return true iff point p is in or on the rectangle
    public boolean encloses( Point p ) { … }
}
```

Depict a thorough set of test case equivalence classes for the `encloses()` method., also indicating the expected results. Add explanatory text as needed for analogous or unusual cases. State your assumptions. Do not implement the method. Do not write test code. Do not use JUnit.


false
true
true
false
false
false
true
true
(—,—)
(—,—)


false
false
true
true
true
..


false
false
true
....


false
true

- 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.

```
class MockLocation extends Location {
    boolean setterCalled;

    public MockLocation( String latitude, String longitude ) {
        super( latitude, longitude );
        setterCalled = false;
    }

    public void setLatitude( String latitude ) {
        super.setLatitude( latitude );
        setterCalled = true;
    }

    public void setLongitude( String longitude ) {
        super.setLongitude( longitude );
        setterCalled = true;
    }

    public boolean getSetterCalled() {
        return setterCalled;
    }
}
```

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

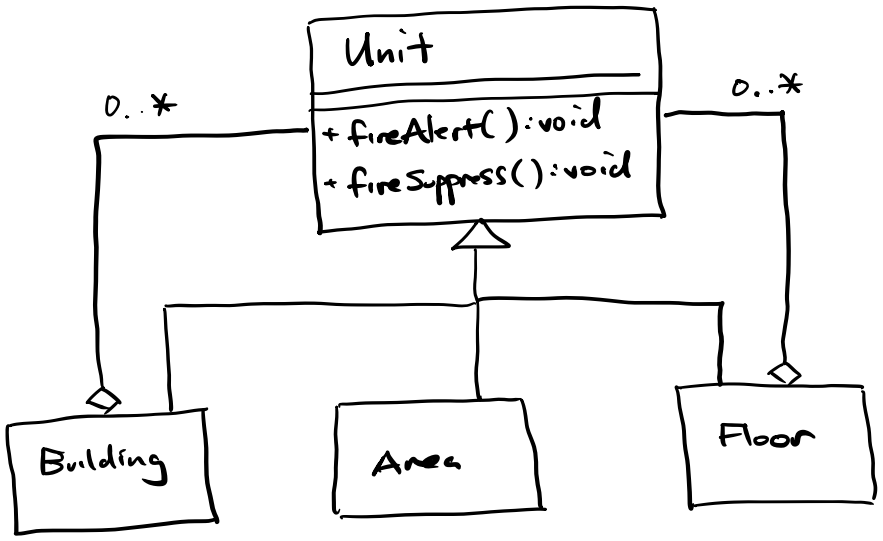
```
class TestTrackAdd extends TestCase {
    public void testNoLocationSetter() {
        Track track = new Track();


        MockLocation location = new MockLocation( "53 N", "113 W" );
        track.add( location );
        assert !location.getSetterCalled();









    }
}
```

- Design patterns

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

[3] 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.



assume some other code assembles the objects appropriately, so that a building contains floors and a floor contains areas

To represent the hierarchical structure of this building (with floors and areas), and support consistent fire alert and suppression behavior across these units, the composite design pattern (tree structure and uniform behavior) is appropriate here.

- 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)?

```
When an observer object is asked to update itself, it can be passed the
particular subject object that had changed.

So, rather than just update() in the observer, change the method to
update( Subject ).
```

(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.

```
Just prior to deleting the subject, each observer object should be
notified of that, and do what's appropriate.

So, add a detach() method in each observer to be called just prior to
the actual deletion of the subject.
```

- 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.

```
You have a hierarchical structure of directories and contents, and
uniform behavior in that these things have a size. The composite design
addresses representing hierarchies of things in a uniform way.
```

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

```
You have cells that need to be recalculated when other cells change. A
cell can be notified it needs recalculation using the observer design
pattern. A cell can be considered an observer of the other cells it may
depend on.
```

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

```
You have the functionality of a college math calculator, but need to be
an interface a normal kid could use. So, you are providing a new,
simplified interface over a more complex system, so the facade design
pattern is appropriate here.
```

(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.

```
You have different combinations of file filters (compression,
encryption) that need to be assembled to handle the input file
appropriate. The decorator pattern supports constructing these
combinations and processing the input, where each decorator is a
particular type of file filter.
```

- 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.

The main code smell is the switch statement conditional on the pay type. Rather than such conditionals, we should use specific types of objects that do the right thing for each case.

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

One approach is to refactor the code to use the state design pattern, with pay type subclasses for EngineerPayType, SalesPersonPayType, and ManagerPayType, and a payAmount() method in each that returns the appropriate pay.

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