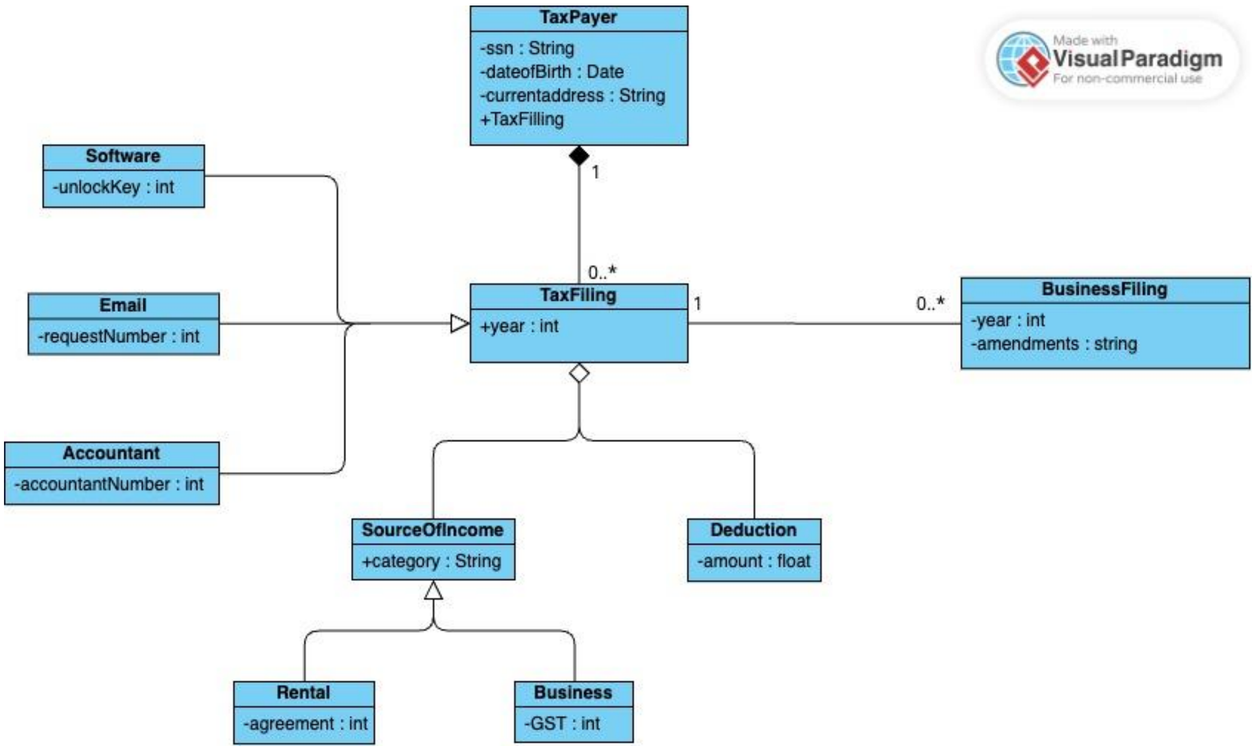
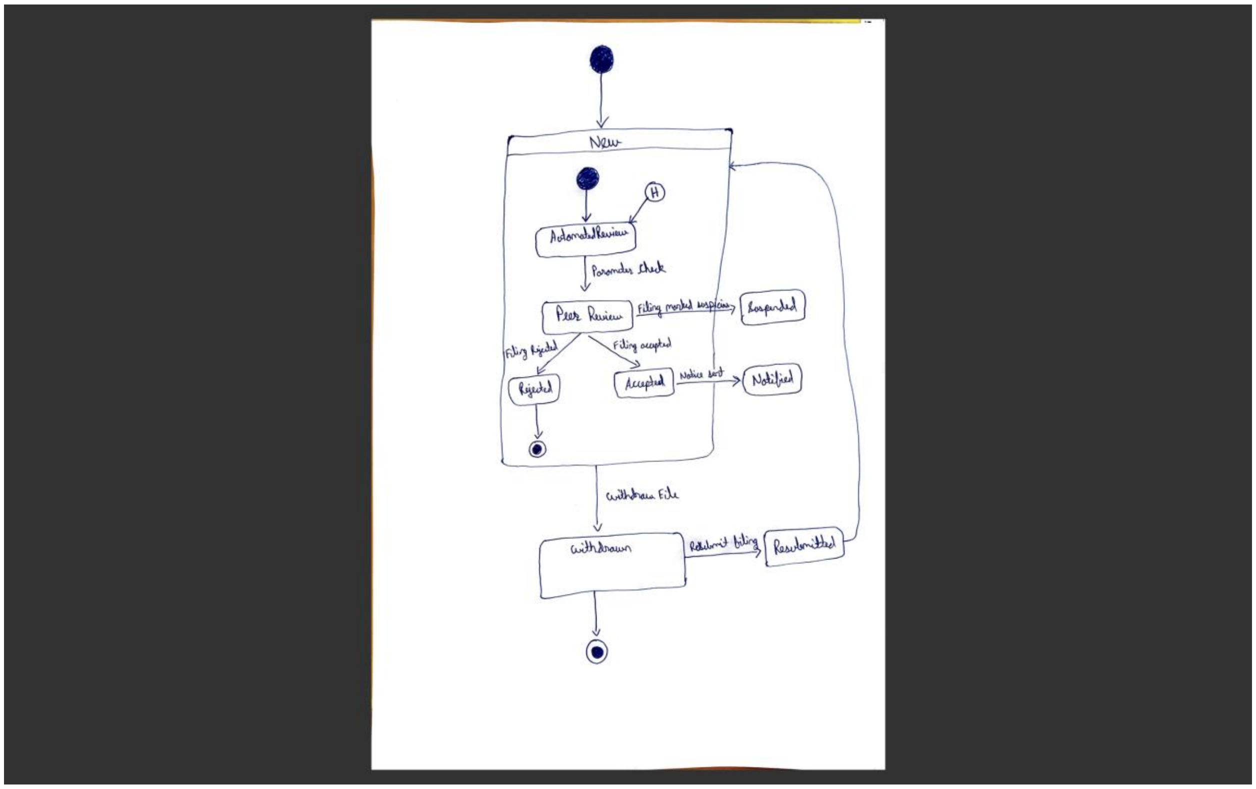
Siddharth Bahekar – 002417718

Q1)



Q2)


New
H
AutomatedReview
Paramater check
Peer Review
Filing marked suspicious
Suspended
Filing Rejected
Filing accepted
Rejected
Accepted
Notice sent
Notified
withdraw File
withdrawn
Resubmit filing
Resubmitted

Q3)

```
 interface Filter {
    + filter(filings: Set<Filing>): Set<Filing>
}

class AndFilter implements Filter {
    - filters: List<Filter>

    + filter(filings: Set<Filing>): Set<Filing>
}

class OrFilter implements Filter {
    - filters: List<Filter>

    + filter(filings: Set<Filing>): Set<Filing>
}

class NotFilter implements Filter {
    - filter: Filter

    + filter(filings: Set<Filing>): Set<Filing>
}

class FilingTypeFilter implements Filter {
    - filingType: FilingType

    + filter(filings: Set<Filing>): Set<Filing>
}

class SubmissionTypeFilter implements Filter {
    - submissionType: SubmissionType

    + filter(filings: Set<Filing>): Set<Filing>
}

class SuspiciousFilingFilter implements Filter {
    + filter(filings: Set<Filing>): Set<Filing>
}

class LateFilingFilter implements Filter {
    + filter(filings: Set<Filing>): Set<Filing>
}

class PriorSuspiciousFilingFilter implements Filter {
    + filter(filings: Set<Filing>): Set<Filing>
}
```

Q4)

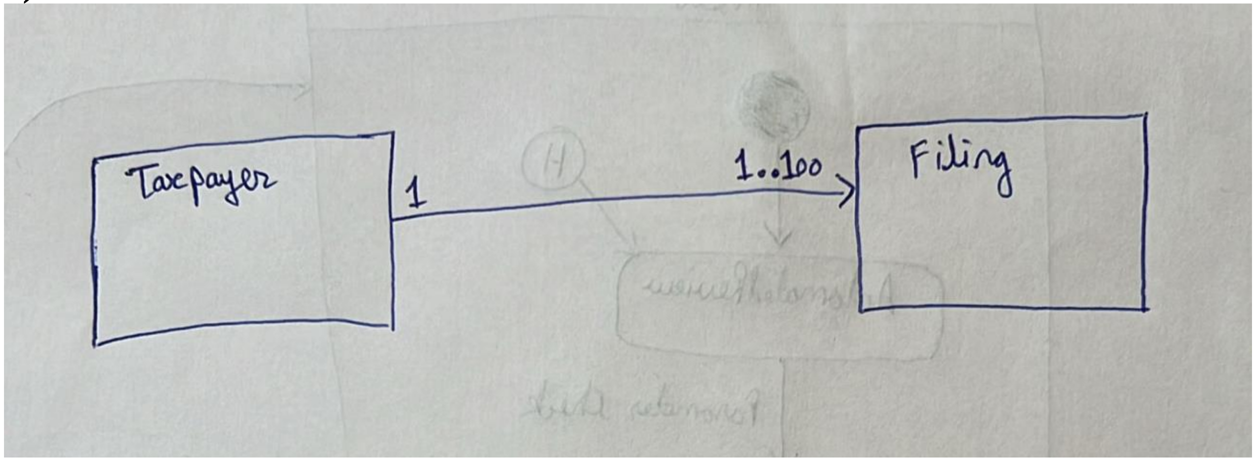
1)

```
@author sid
public class TaxPayer {

        private String name;

        private String phoneNumber;

       private String emailAddress;

    /**
     * Submits a filing for the taxpayer.
     *
     * @return true if the filing was successfully submitted, false otherwise.
     */
    public boolean submitFiling() {
        // implementation here
    }

    /**
     * Views the notice of assessment for the taxpayer.
     *
     * @return true if the notice of assessment was successfully viewed, false
otherwise.
     */
public boolean viewNoticeOfAssessment() {
        // implementation here
    }

    /**
     * Gets the email address of the taxpayer.
     *
     * @return the email address of the taxpayer.
     */
    public String getEmailAddress() {
        // implementation here
    }

    /**
     * Gets the phone number of the taxpayer.
     *
     * @return the phone number of the taxpayer.
     */
    public String getPhoneNumber() {
        // implementation here
    }

    /**
     * Sets the name of the taxpayer.
     *
     * @param name the name of the taxpayer.
     */
    public void setName(String name) {
        // implementation here
    }

    /**
     * Checks if the taxpayer has any outstanding filings.
     *
     * @return true if the taxpayer has any outstanding filings, false otherwise.
     */
    public boolean hasOutstandingFilings() {
        // implementation here
    }

    /**
     * Checks if the taxpayer has any prior suspicious filings.
     *
     * @return true if the taxpayer has any prior suspicious filings, false otherwise.
     */
    public boolean hasPriorSuspiciousFilings() {
        // implementation here
    }
}
```

2)

```
 public class TaxPayerTest {

    private TaxPayer taxPayer;
    @BeforeEach
    public void setUp() {
        taxPayer = new TaxPayer();
        taxPayer.setName("Sid Max");
        taxPayer.setEmailAddress("sidmax@example.com");
        taxPayer.setPhoneNumber("111-1111");
    }

    @Test
    public void SubmitFiling_ test () {
        boolean result = taxPayer.submitFiling();
        assertTrue(result);
    }

    @Test
    public void ViewNoticeOfAssessment_ test() {
        boolean result = taxPayer.viewNoticeOfAssessment();
        assertTrue(result);
    }

    @Test
    public void GetEmailAddress_ test () {
        String result = taxPayer.getEmailAddress();
        assertEquals("sidmax@example.com", result);
    }

    @Test
    public void GetPhoneNumber_ test () {
        String result = taxPayer.getPhoneNumber();
        assertEquals("111-1111", result);
    }

    @Test
    public void SetName_ test () {
        taxPayer.setName("Sid Max");
        String result = taxPayer.getName();
        assertEquals("Sid Max", result);
    }
}
```

Q5)
1)
The alternative A is the correct as per the requirement statement. The question statement can be broken into three pieces and alternative A satisfies it -
TaxPayer can view a list of filings : The association between the classes depicts the relation and says these two classes are linked.
But the filing class does not link to the taxpayer – This is where the unidirectional arrow from TaxPayer to Filing satisfies the condition.
Every taxpayer can be associated with many filings – The multiplicity is correct, each taxpayer can be associated with many filings

2)


Q6)
1)
Multithreading is complex because we have multiple actions fighting for the same resource. Imagine two chefs working in a kitchen with only one cutting board and knife. Each chef needs to prepare their own dishes, which involve chopping, slicing, and dicing. However, they both occasionally need to use the knife.

When one chef is using the knife, the other has to wait. If the knife becomes available, the other chef can then use it, but they must wait their turn if the first chef needs it again. Their individual tasks slow down slightly, together they can accomplish more than one chef alone.

There is also the issue of a race condition. If the knife and cutting board are both free, and one chef grabs the knife while the other snatches the cutting board, neither is willing to give up their resource, leading to a deadlock. You would find two frustrated chefs with half-prepared dishes. This race condition is a primary challenge in multithreading.

A similar scenario might occur in an art studio with multiple painters sharing one palette of colors. If one painter wants to refill a color while another needs to use it, chaos could ensue, resulting in spilled paint and messy artwork. Again, protecting the resource with a Mutex-like mechanism ensures smooth cooperation among the painters.

I have tried to cover 3-4 difficulties faced in multithreading as compared to single threaded program, they are, concurrency, non-determinism, complexity and competition for resource.

2)

In Java, synchronization is used when we need only one thread to access a specific resource at a time. If multiple threads try to access the same resource simultaneously, synchronization ensures that only the first thread gets access. After the first thread is done, then the next one can use the resource.

We use synchronization in Java to prevent deadlocks. A deadlock happens when two threads are both trying to access the same resource and they get stuck, so neither can finish.

Think of it like when a few people want to print documents from one printer. If there's no system to manage it, the printouts could get mixed up. Similarly, in programming, there are times when only one process should work with data at a time. For example, if someone is transferring money from both a website and a mobile app simultaneously, the process needs to happen one after the other to avoid problems.

To handle this, we wrap important actions in a synchronized block. This makes sure that if multiple processes are trying to do the same thing, they have to wait in line and do it one by one.

3)

Supportive : In Java and Object-oriented programming, design patterns play a crucial role in creating robust and maintainable software. They provide reusable solutions to common problems encountered during software development. Therefore, when the text mentions "Designs like the," it could be interpreted as a reference to established design patterns such as Singleton, Factory, or Observer. These design patterns encapsulate best practices and promote code reusability, scalability, and maintainability. By utilizing these designs effectively, developers can streamline development processes and produce high-quality software.

Critic : Design patterns are useful in Java and Object-oriented programming, using them without fully understanding a project's specific needs can make things more complicated than necessary. When we hear "Designs like the," it might imply a one-size-fits-all approach, ignoring the importance of tailoring solutions to each project's unique requirements. Relying solely on design patterns can also reduce creativity and innovation by limiting exploration of other potentially better-suited solutions. So, while design patterns are helpful, developers should carefully assess their suitability for each situation to avoid adding unnecessary complexity and inflexibility to the software architecture.

References -

ChatGPT
Google
Quora