Instructions TL;DR:

* Please completely fill in circles on the marking page. No external aids are permitted.
* NOTHING in this booklet will be marked, just the marking page (with the bubbles).
* Negative marks do not flow across LETTERED questions.
* Correct: +2, Incorrect: -1; Blank: 0.
* Exam is 75 minutes. Do not leave in first 20 or last 20.

Detailed instructions:

All questions on the exam are true/false. Every lettered question will have zero or more true answers. Exam was designed for 50 minutes, but you can take up to 75 minutes.

Correct answers are awarded 2 marks.
Incorrect answers are awarded -1 mark.
Blank answers receive no marks or penalties.

The minimum grade on any lettered question (e.g., A), B), etc.) is 0; negative marks will *not* carry across lettered questions (but will carryover between *numbered* sub-questions within a *lettered* question). For example, 5 / 5 right answers for a lettered question is worth 10 marks; 4 / 5 is worth 7 (or 8 if the 5th was blank).

A)

1. Measuring specification adherence is typically straightforward.
2. JavaScript callbacks usually use an error-last idiom.
3. Promises make nested callback chains perform slower at runtime.
4. Software testing can show the presence but not the absence of bugs.
5. White box testing is more widely used in practice than black box testing.

B)

6. Complexity is a dominant concern for software engineers.
7. APIs should not hide details of their internal implementation.
8. APIs are harder to misuse if they provide effective affordances.
9. Bottom-up decomposition is the process of breaking up features into methods.
10. Applying the interface segregation principle to an existing system increases the total number of interfaces in a system.

C) While waterfall-based processes are often derided, they do have some advantages.

11. The process is easily understood.

12. Requirements can be easily revisited and improved.

13. Handoffs between stakeholders are explicit and detailed.

14. Encourages decisions that are responsive to organizational change.

D) There are many flavours of agile methodologies; at their core, agile proponents:

15. Believe that teams should be flexible to external change.

16. Think customers should be actively and continually involved.

17. Understand that building the wrong thing is a fundamental risk.

18. Keeping software buildable encourages experimentation and prototyping.

E) Red-green-refactor is an important idiom that:

19. Only applies to TDD.

20. Ensures that tests can fail before a bug is fixed or a new feature is added.

21. Makes it easier to refactor a system.

F) Scrum-based teams:

22. Use the product backlog to track issues that will be fixed in a sprint.

23. Meet only at the end of sprints to perform end-of-sprint ceremonies.

24. Reflect on their process and how it can be improved after each sprint.

25. Continually engage product owners to ensure the right items are being addressed in each sprint.

```
class ElevationController implements BarometricElevation {
    private correction: number; // correct for on-device error
    // returns elevation in meters for a pressure in millibars
    public getElevation(pressure: number): number {
        let altpress = (1 - Math.pow((pressure / 1013), 0.19)) * 145366;
        altpress = (0.305 * altpress)  + this.correction;
        if (altpress < 0) { UI.showWarning('Elevation incorrect'); }
        return altpress;
    }
}
```

Figure B

G) Considering the code in Figure B with respect to the SOLID design *principles:*

26. Given its current design, does `ElevationController` *make it impossible for a* client to use the dependency inversion principle.

27. The `getElevation(..)` implementation adheres to the *single responsibility* principle.

28. Assuming `BarometricElevation` declares only `getElevation(..)`, *the* interface segregation principle is likely being followed.

29. Using the following code in a client adheres to the Open/Closed principle:

```
const ele:ElevationController = new ElevationController();
```

30. Clients would be more flexible if they instantiated a `BarometicElevation` by

```
const ele:BarometricElevation = ElevationFactory.getController();
```

31. By including the `BarometricElevation` declaration in the client package, the code above would adhere to the dependency inversion principle.

H) In terms of the testability of the code in Figure B:

32. All side effects of `ElevationController::getElevation(..)` are observable.

33. `getElevation(..)` is not isolateable because there is not a mechanism for checking problems induced by an incorrect `correction` value.

34. Controllability is a concern because `showWarning(..)` cannot be directed at a different implementation.

35. The intended behaviour of `BarometricElevation::getElevation(..)` is observable.

I) The representation in Figure A is used to:

36. Describe system structure.

37. Reason about deployment.

38. Identify all system classes.

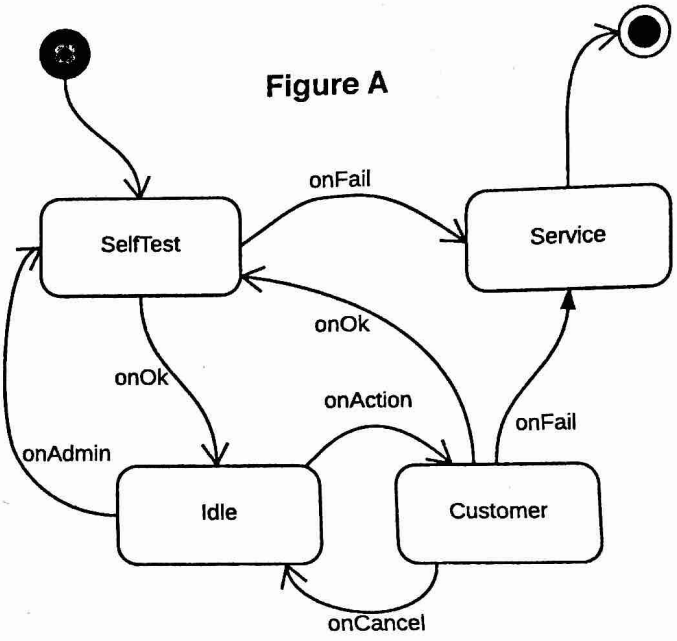

Figure A

J) Figure A describes:

39. The states the system can be in.

40. The events that occur at runtime.

41. All possible system interactions.

K) Coverage-based metrics:

42. Are commonly used in conjunction with black-box testing.

43. Ensures that code is correct.

44. Provide insight into parts of a system that are not tested.

45. Are not computationally expensive to collect.

L) Assertions should validate:

46. That every possible value that could be sent to an API are correct.

47. That some expected values are correct.

48. That some boundary values are correct.

49. That all parts of the code are executed.

50. That our expected values match the specification.