COMP2511

WEEK 3

UML DIAGRAM 😱

IMPORTANT NOTICE

- Assignment-i is out, please start early!!!
- Go to help sessions (go now while they are less busy)
- No auto generated UMLs. You will get 0.

AGENDA

- Design by Contract
- Domain Modelling
- Wondrous (IDE Programming, Writing Tests with JUnit, Exceptional Conditions)

DESIGN BY CONTRACT

- What is Design by Contract?
  - Providing an interface for others to use with clear preconditions, postconditions and invariants which, when adhered to, guarantees correct and expected behaviour.

DESIGN BY CONTRACT

- What is Design by Contract?
    - Providing an interface for others to use with clear preconditions, postconditions and invariants which, when adhered to, guarantees correct and expected behaviour.
- Discuss what preconditions, postconditions and invariants are.
    - Preconditions - conditions on the inputs which guarantee postconditions will be true
    - Postconditions - guarantees from the actual software on what you can expect from a function (given preconditions)
    - Invariants - guarantees from the actual software that are always maintained before and after a function call (they may not always hold during the function call, but the user shouldn't need to worry about that)

DESIGN BY CONTRACT

- In the people package, there are a few classes which represent the people at a university - Briefly discuss the preconditions and postconditions of the constructors, getters and setters in Person.java - Fill in the preconditions and postconditions for setSalary in Person.java - Discuss the validity of the subclasses of Person, and why they are/aren't valid subclasses - Fix any issues you identified before.

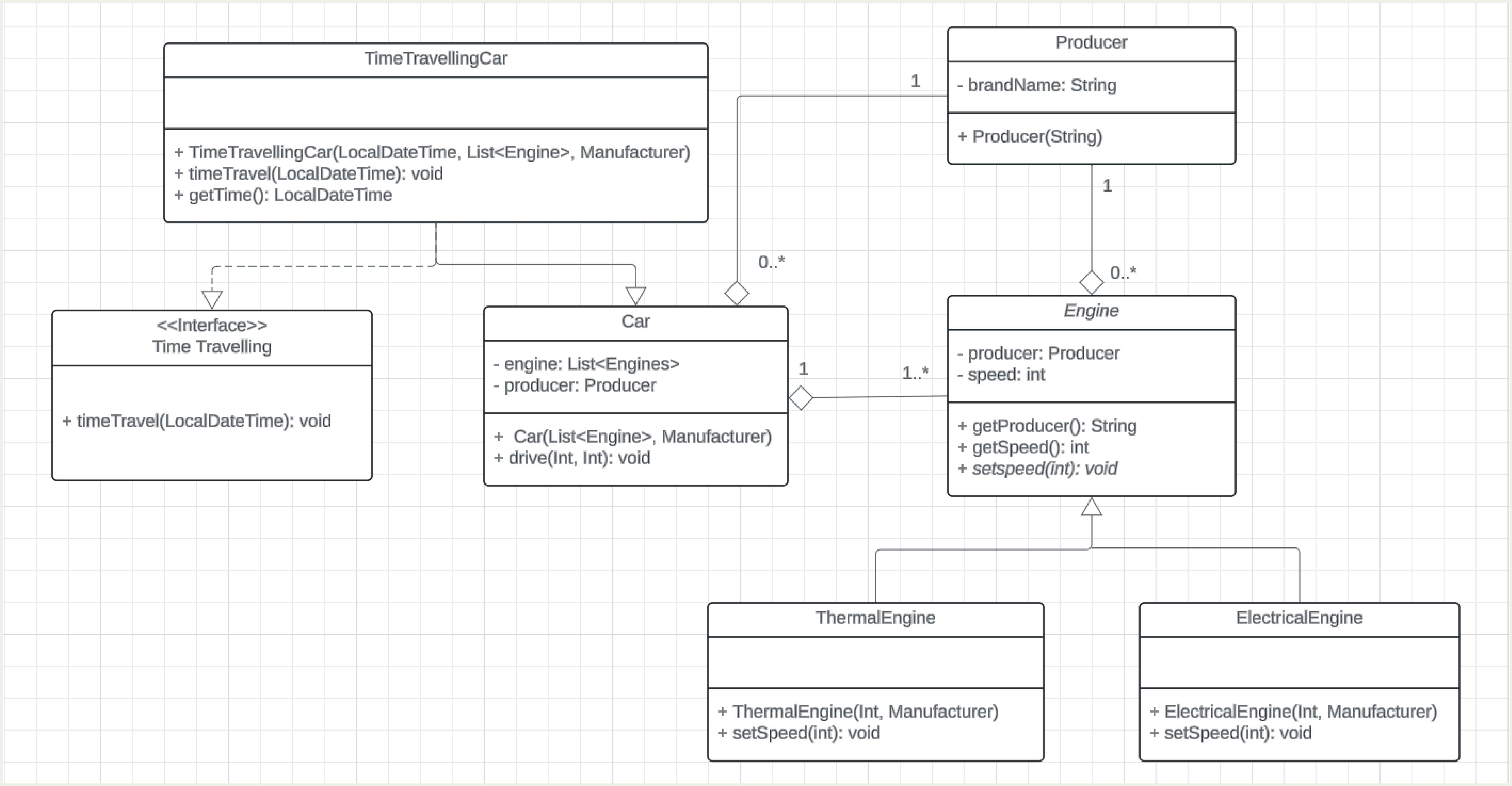
DESIGN BY CONTRACT

- Will you need to write unit tests for something that doesn't meet the preconditions? Explain why.
    - No, as the point is that inputs which don't meet the preconditions are not accounted for

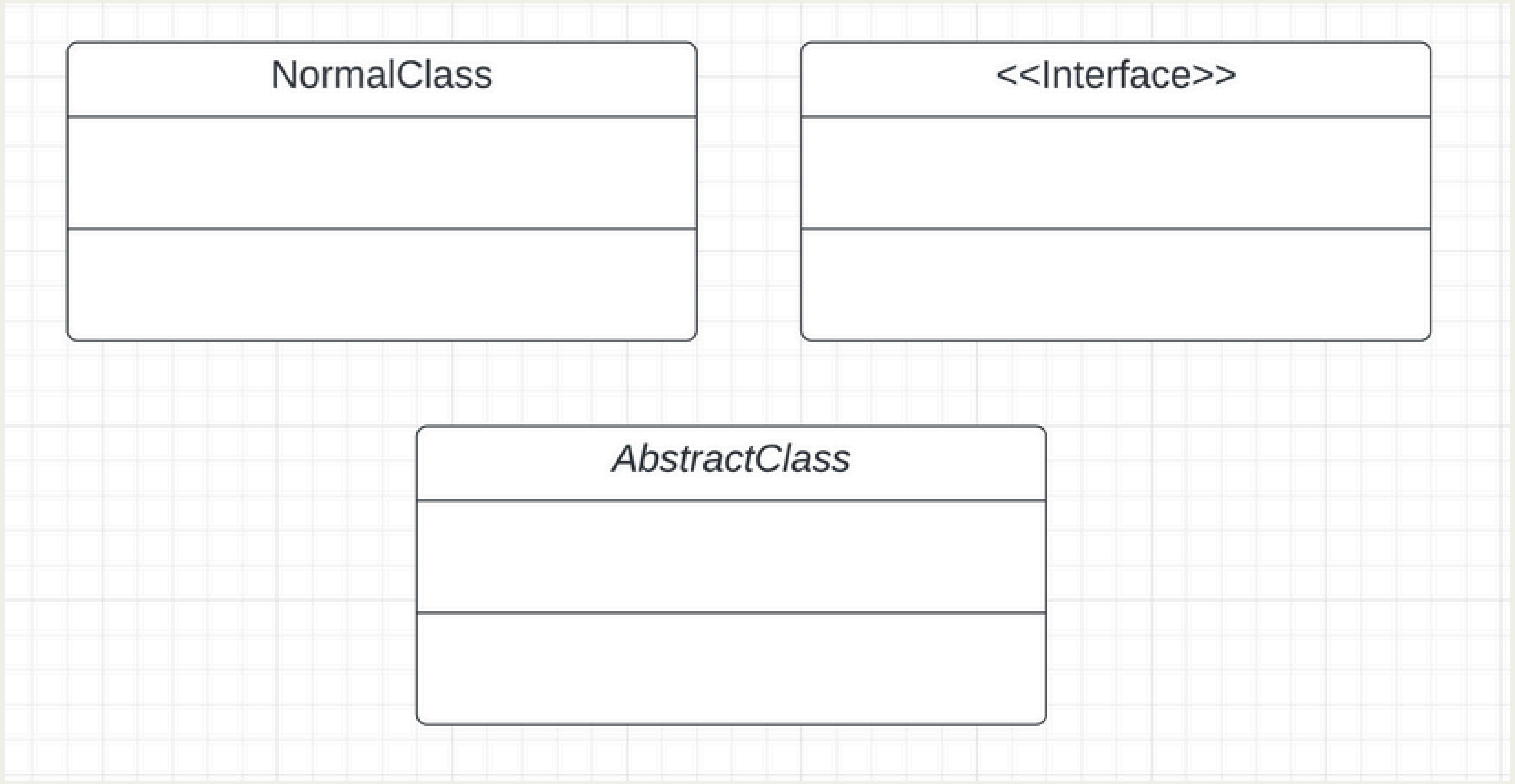
DOMAIN MODELLING

- Domain Models are used to visually represent important domain concepts and relationships between them.
- We will use Unified Modeling Language (UML) class diagrams to represent domain models.
- There will most likely be a domain modelling question in your exam.

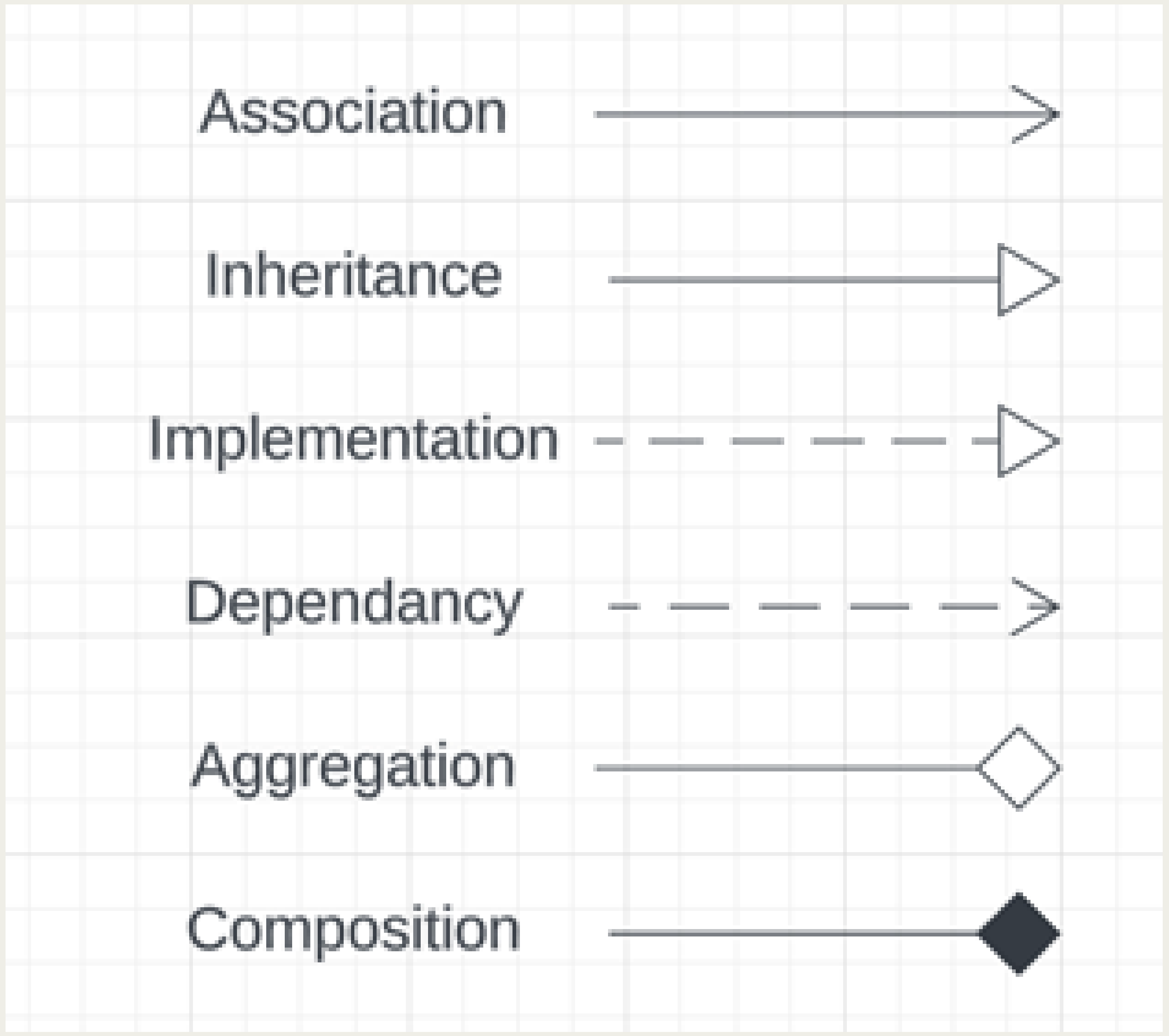
UML EXAMPLE


TimeTravellingCar
+ TimeTravellingCar(LocalDateTime, List<Engine>, Manufacturer)
+ timeTravel(LocalDateTime): void
+ getTime(): LocalDateTime
Producer
- brandName: String
+ Producer(String)
1
1
0..*
0..*
<<Interface>>
Time Travelling
+ timeTravel(LocalDateTime): void
Car
- engine: List<Engines>
- producer: Producer
+ Car(List<Engine>, Manufacturer)
+ drive(Int, Int): void
1
1..*
Engine
- producer: Producer
- speed: int
+ getProducer(): String
+ getSpeed(): int
+ setspeed(int): void
ThermalEngine
+ ThermalEngine(Int, Manufacturer)
+ setSpeed(int): void
ElectricalEngine
+ ElectricalEngine(Int, Manufacturer)
+ setSpeed(int): void

UML DIAGRAM - TYPES OF CLASSES



UML DIAGRAMS- RELATIONSHIPS



Association: a class uses another class in some way.

Inheritance: a class inherits another class.

Implementation: a class implements an interface.

Dependency: a class depends on another class.

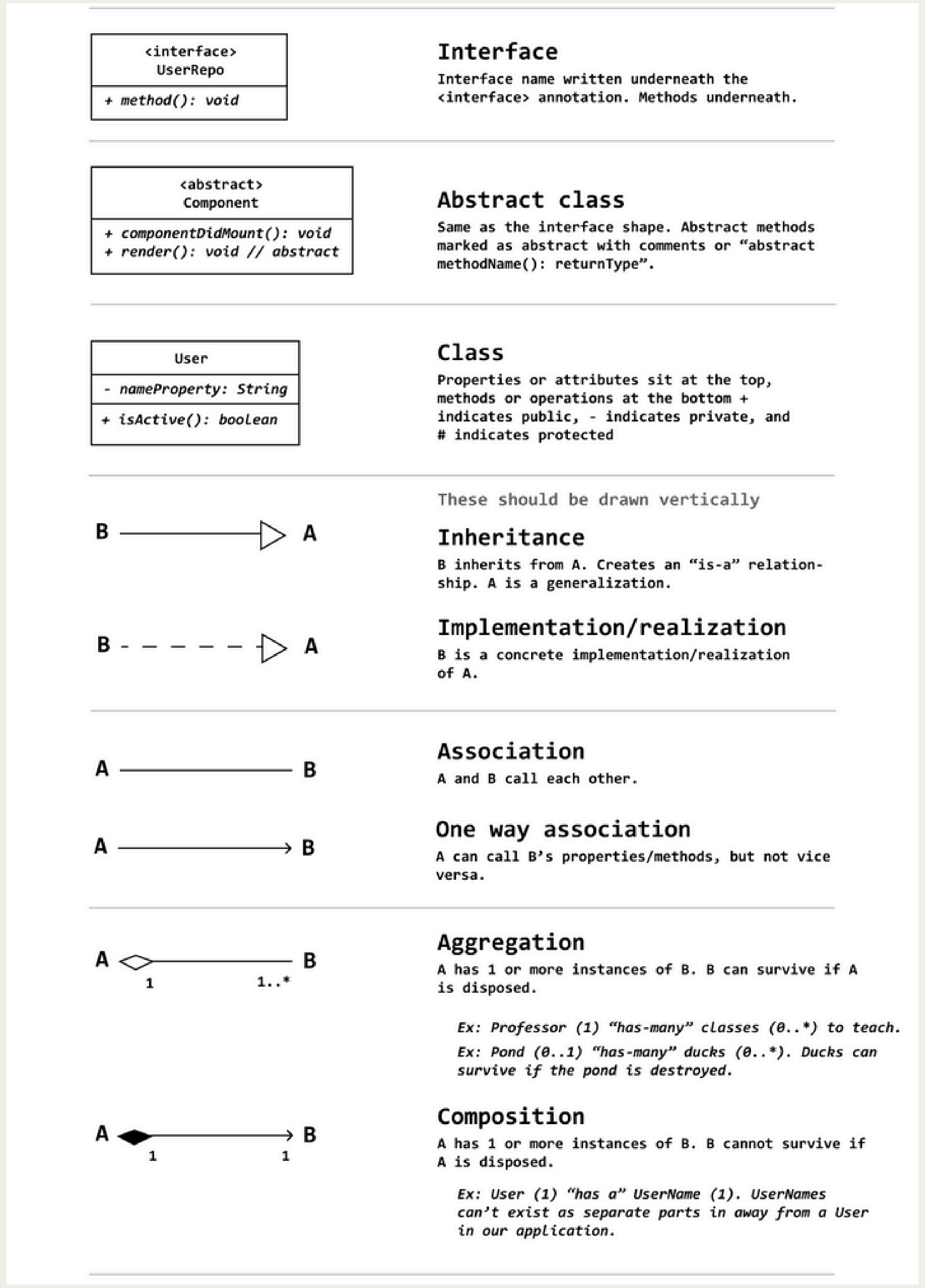
Aggregation ("has-a" relationship): a class "A" contains another class "B". "B" **can** exist independently of "A".

Composition ("has-a" relationship): a class "A" contains another class "B". "B" **cannot** exist independently of "A".

UML DIAGRAMS - COMPOSITION

Composition is a specialized form of aggregation. In composition, if the parent object is destroyed, then the child objects also cease to exist. Composition is actually a strong type of aggregation and is sometimes referred to as a "death" relationship.

UML DIAGRAMS - CHEAT SHEET



```
<interface>
UserRepo
+ method(): void
```

Interface

Interface name written underneath the <interface> annotation. Methods underneath.

```
<abstract>
Component
+ componentDidMount(): void
+ render(): void // abstract
```

Abstract class

Same as the interface shape. Abstract methods marked as abstract with comments or “abstract methodName(): returnType”.

```
User
- nameProperty: String
+ isActive(): boolean
```

Class

Properties or attributes sit at the top, methods or operations at the bottom + indicates public, - indicates private, and # indicates protected

These should be drawn vertically

$B \longrightarrow A$

Inheritance

B inherits from A. Creates an “is-a” relationship. A is a generalization.

$B \dashrightarrow A$

Implementation/realization

B is a concrete implementation/realization of A.

$A \text{———} B$

Association

A and B call each other.

$A \rightarrow B$

One way association

A can call B’s properties/methods, but not vice versa.

Aggregation

A has 1 or more instances of B. B can survive if A is disposed.

Ex: Professor (1) “has-many” classes (0..) to teach.*

Ex: Pond (0..1) “has-many” ducks (0..). Ducks can survive if the pond is destroyed.*

Composition

A has 1 or more instances of B. B cannot survive if A is disposed.

Ex: User (1) “has a” UserName (1). UserNames can’t exist as separate parts in away from a User in our application.

DOMAIN MODELLING

Create an OO domain model for a system with the following requirements.

A Car has one or more engines and a producer. The producer is a manufacturing company who has a brand name. Engines are produced by a manufacturer and **have a speed**. There are only two types of engines within UNSW's cars:

- **Thermal Engines**, which have a default max speed of 114, although they can be produced with a different max speed, and the max speed can change to any value between 100 and 250.
- **Electrical Engines**, which have a default max speed of 180. This is the speed at which they are produced, and the max speed can change to any value that is divisible by 6.

Cars are able to drive to a particular location x, y.

Since UNSW is a world-leader in technology innovation, they want you to be able to model the behaviour of Time Travelling for *any* vehicle, and to model a time travelling car. A vehicle that travels in time *stays in the same location* but travels to a LocalDateTime.

DOMAIN MODELLING

Create an OO domain model for a system with the following requirements.

A Car has one or more engines and a producer. The producer is a manufacturing company who has a brand name. Engines are produced by a manufacturer and **have a speed**. There are only two types of engines within UNSW's cars:

- **Thermal Engines**, which have a default max speed of 114, although they can be produced with a different max speed, and the max speed can change to any value between 100 and 250.
- **Electrical Engines**, which have a default max speed of 180. This is the speed at which they are produced, and the max speed can change to any value that is divisible by 6.

Cars are able to drive to a particular location x, y.

Since UNSW is a world-leader in technology innovation, they want you to be able to model the behaviour of Time Travelling for *any* vehicle, and to model a time travelling car. A vehicle that travels in time *stays in the same location* but travels to a LocalDateTime.

WONDROUS

The **Wondrous Sequence** is generated by the simple rule:

- If the current term is even, the next term is half the current term.
- If the current term is odd, the next term is three times the current term, plus 1.

For example, the sequence generated by starting with 3 is:

```
3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
```

If the starting term is 1, then an empty list is returned.

Code Demo

Part 1 - IDE Programming (VsCode Debug Mode)

Part 2 - Writing Tests with JUnit

Part 3- Exceptional Conditions

IDE Programming

1. Put a breakpoint on line 13 and run the tests in Debug Mode.
2. Briefly discuss different features of Debug Mode:
    a. The variables section
    b. The debug console
    c. The 'watch' section
    d. The call stack
    e. Debug control
3. Use the debug tools and the given algorithm to determine why the test is failing, and fix the bug.

Writing Tests with JUnit

There is a further bug in the function not caught by the given unit test. Find the other bug, and write a corresponding unit test inside WondrousTest.

You can learn more about JUnit here.

Exceptional Conditions

Modify the method such that if start is less than 1, an IllegalArgumentException is thrown. Write a corresponding test for this inside WondrousTest.

In many cases when we throw an exception we need to update the method signature and existing tests but here we don't - why is this?

Exceptions

- An exception is an event, which occurs during the execution of a problem, that disrupt the normal flow of the program's instructions
- When error occurs, an exception object is created and given to the runtime system. This is called throwing an exception
- The exception handler chosen is said to catch the exception

Exceptions-Checked vs Unchecked

- **Checked**: Must be checked, will result in compilation error if not handled
  - Any class that inherits from `Exception` is a checked exception.
  - E.g., IOException, SQLException
- **Unchecked**: Genuine errors that occur at run time
  - Any class that inherits from `RuntimeException` is unchecked
  - E.g., ArrayIndexOutOfBoundsExceptions, ArithmeticException