Pleasure and Perils of Inheritance

- Inheritance is widely used. Often, it is overused.

Relationships between Abstractions

- Abstractions are representations of ideas, concepts. Even tangible and intangible things are represented in abstractions.
- Abstractions rarely live in isolation, often relate to other abstractions. This is called **coupling**. Coupling needs to be minimised as much as possible. A system with no coupling would probably not exist at all.
- **Association** → relationship between two objects at the simplest form. Stronger relationship would be **Aggregation** ⇒ one object becomes a part of another object.
    - Eg. A car has an engine (aggregation, part-whole relationship, ‘has a’ relationship).
    - Eg. Joe & Sam are friends (association, equal status).
- ‘Is a’ or ‘kind of’ relationship is *most likely* an inheritance. Eg. Car *is a* vehicle. This doesn’t necessarily imply inheritance.
    - Better ask “Can I use a Car anywhere I can use a Vehicle?” “Yes”→ this is inheritance model. “Can I use a Car anywhere I can use an Engine?” “No”→ Not an inheritance relationship.

Inheritance

What’s that?

When a particular abstraction brings along all the details of another abstraction and builds on top of that, it is called **inheritance**.

Example→

We have a class called `Vehicle`, with a `drive` method. We have another class called `Car`. This `Car` class inherits from the `Vehicle` class. `Car` *is a* `Vehicle` → inheritance.

We can use a `Car` object anywhere a `Vehicle` object is expected. Due to inheritance, `Car` automatically obtained all the properties of `Vehicle`, so as a result, the `drive` method becomes available in the `Car` as much as it is in `Vehicle`.

Main Purpose

The main purpose of inheritance is to reuse an abstraction. However, inheritance also increases coupling — double-edged sword. Since the class depends on a concrete implementation of another class, the former gets affected in the case that the latter is modified → worst form of coupling.

Benefits and Burdens

Inheritance increases the burden of coupling → bad.

When a class is extended (inheritance), the derived class/implementation has to bind by the contractual expectations of the base class. The contractual expectations cannot be violated (cannot go ahead do something wildly different). The base class imposes certain expectations on the derived class, because of **substitutability**.

The derived class can reimplement a method, but the external behaviour has to be conformant with the external behaviour of what the base class provides.

Inheritances increases the burden of the contractual obligation. It is not a reuse that comes for free. It is a reuse that imposes a contract/constraint on the implementor which receives the reuse.

So, when `Car` inherits from `Vehicle`, `Vehicle` imposes certain expectations on the `Car` in terms of the substitutability.

Interface vs. Implementation Inheritance

- **Interface Inheritance is much better than Implementation Inheritance.**
- Eg. In Java, `List` is an interface, while `ArrayList` or `LinkedList` are implementations.

- Interface gives a contract without any implementations details. You get a looser coupling.
    - An interface has certain expectations too.
        - The methods of an interface may still have a particular contract even though they do not have an implementation.
        - Eg. If `Vehicle` were an interface, the `drive` method still has certain connotation to it. `drive` expects that the distance travelled should increase or that the vehicle moves in a certain direction.
- Implementation Inheritance give a stronger coupling.
- We prefer Interface Inheritance/loose coupling over Implementation Inheritance/strong coupling, because loose coupling is more extensible than strong coupling.
- We cannot create instances of interfaces. However, we can treat instances of other concrete classes as if they were an implementation of the interface.
    - Eg. If `Vehicle` were an interface, we cannot create an instance of `Vehicle`, however, we can treat an instance of `Car` as if it were an instance of `Vehicle`.
- Languages differ in how they offer inheritance.
    - In a language like Java, you're required to implement all the methods in the interface; No concept of optional methods. A language like Objective C allows partial inheritance, with required methods and optional methods.
    - Generally, you can only implement an interface once in a class.
        - However, in C#, you can provide a couple of different implementations of the same interface in a class; One could be a simple inheritance of the interface; You can also do implicit inheritance of interface.
            - Implicit Inheritance in C# → you can go to an object and ask for a separate implementation of a specific interface.

Inheritance vs. Delegation & When to use Inheritance

- Inheritance gives us reuse. However, reuse can be obtained not just from inheritance but from other forms as well.

- If a class B should be used anywhere an object of a class A is used, then use inheritance. Otherwise, for reuse, use **delegation**.
  If an object of B should use an implementation of A, then use delegation.
    - “Do you want to use an abstraction or do you want to be used as an abstraction?”
        - If you want to use an abstraction, use delegation.
        - If you want to be used an abstraction, use inheritance.
- Delegation is better to use than inheritance, but many languages don’t even offer delegation.
    - Because many languages do not even offer delegation, inheritance is overused.
- **Inheritance is very static** in nature. When you inherit a class from another class, it is static. The inheritance is bound at the compile time.
- **Delegation is dynamic**. You can point to an object now and change to another at a later time.
    - Dynamic behaviour is better because it give more extensibility and flexibility.
- Using Inheritance or Delegation — need to make this decision based on the usage, not just the hypothetical relationship.
- `Groovy` language has a `@Delegate`, so even if you add new methods, you don’t need to make any changes to the class using the delegate, that is, no violation of OCP.
    - In some other language, you will have to manually add the new methods.