the tedium of coding masses of identical code for each type of aeroplane. The first class `aeroplane` is the one which contains the data and methods we can share amongst all aeroplanes. Since it is only a template, we designate it `abstract`. We will never instantiate an object of type `aeroplane`, only copy it into other objects. This technique is demonstrated in our second and third classes; the effect of the line

```java
class Airliner extends Aeroplane
```

is to give our class `Airliner` access to all the data and methods of our abstract class `Aeroplane`. This means that all we have to do is to give our new class the data that makes it different (i.e. the number of seats) and we have a fully functional airliner. Similarly the third class, `Fighter`, is given guns. It is important to note that `Fighter` would not get the extra seats, unless it extended `Airliner` instead of `Aeroplane`. The inheritance hierarchy in this case looks like Figure 3.1.

![Figure 3.1 Inheritance from superclass Aeroplane](image)

This is a very simplistic example of inheritance, and assumes a great deal – for instance where did the gun come from? How does the engine work? The answers to these questions require a section to themselves.

### 3.2 Introduction to object orientation

In order to understand object orientation fully it is necessary to take a step backwards and examine the history of software engineering and the way in which it is maturing as an industry.

Software engineering is a very youthful industry, and we can look to other industries for possible reflections and clues as to what may happen in the future. If we look at the oldest industry of all, that of construction, we see an interesting phenomenon. That phenomenon is specialization and it is this that enables us to erect structures such as St Paul’s Cathedral from scratch, and then to refine the techniques to build bigger and better versions.

If we look more closely at specialization, we can see some ideas that we can use in software engineering. The whole point of specialists is that they are the experts – all anyone else needs to know is where to find them and what they might expect them to do. Object orientation attempts to model this relationship. If we can use an existing class as a basis for another class, then we need to know only where to find it and what we can expect it to do. This is referred to as `inheritance` in software engineering. Inheritance is only one of three ways in which classes can be related.

We have looked at an example that models inheritance. This could be referred to as an `ISA` relationship – an airliner `ISA` aeroplane. There is another type of relationship that is not so simply modelled – `HAS A`. It would be unlikely that the good people at Sun have modelled classes to provide us with guns and engines for aeroplanes, and yet these are complex pieces, essential to an aeroplane’s ‘aeroplaneness’. This relationship is termed `composition`. An aeroplane is composed of things of which it categorically is not an example. So, an airliner `ISA` aeroplane and it `HAS A` engine. When designing class structures, it is important to get this relationship right in order to optimize the benefits of object orientation. In Figure 3.2 we see that Object B (Car.java) uses Object A (Engine.java) as a component. Object C (Turbo.java) inherits all the attributes of Object A (Engine.java) and adds some more, such as fuel injection.

![Figure 3.2 Object relationships – inheritance and composition](image)