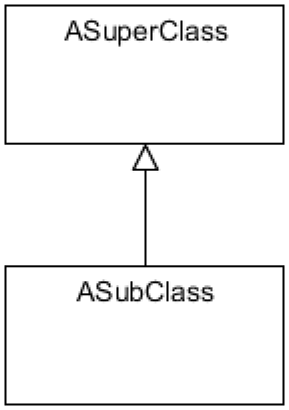
Inheritance and Composition

Inheritance

Picture:



Inheritance is a IS-A relationship.

Another way to think about this is that the superclass is a *generalization* of the subclass and the subclass is a *specialization* of the superclass. So, if Industrial IS-A Zone then we can say that:

- Zone is a generalization of Industrial, and
- Industrial, Powerplant, Residential, etc. are specializations of Zone.

In the latter case, we factor out the common behavior from Industrial, Powerplant, and Residential into the Zone class. Then, having Industrial, etc. inherit from Zone allows us to have that common behavior used in both.

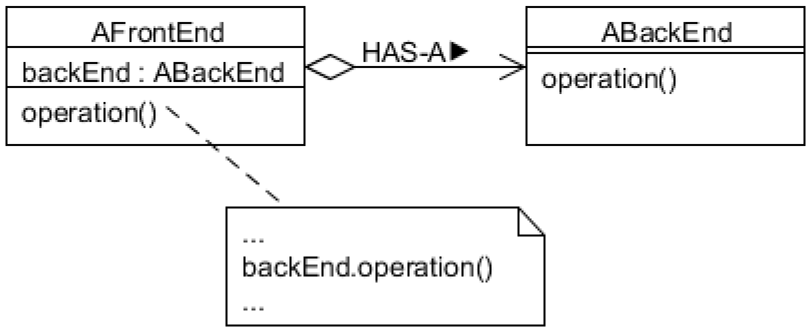
The fact that an Industrial IS-A Zone implies that anything that can be done with a Zone can also be done with an Industrial. For this reason, the behavior of an Industrial must include all the behavior of a Zone. An Industrial object can modify the behavior of a Zone. But, the methods on Industrial should do "the same thing" as the methods on Zone.

If we pursue this idea far enough, there will be cases where a given class could be a specialization of multiple classes which are not themselves related by inheritance. This situation is known as *multiple inheritance*. C++ and Python support multiple inheritance, but, Java (and Ada) do not. The reason for this is that, in general, it is difficult to resolve conflicts in behavior between multiple super-classes. Both Java and Ada use interfaces as a weaker form of multiple inheritance.

When using inheritance the subclass inherits both behavior and implementation—attributes and operations, both internal and external—from the superclass. The subclass' behavior must include all behavior of the superclass. The subclass can override superclass operations and add new operations. However, the subclass must ensure that any overrides are consistent with expectations of the superclass. If the subclass overrides a method that is used internally by the superclass, the subclass must do it in a way that meets the expectations of the superclass. In Java, the superclass can protect itself from having a subclass override an internal operation by making the internal operation private.

Composition

Picture:



Composition is a HAS-A relationship.

In composition, we build complex behavior by having a “front-end” class use one or more “back-end” classes to extend the behavior of the front end. We can think of this as assembling the front-end from parts represented by the back-end(s).

For example, a Car HAS-A Engine. When using composition, it's the responsibility of the front-end class, the Car, to expose whatever subset of the behavior of the back-end class, the Engine, that is part of the behavior of the front-end. For example, starting the car includes starting the engine but can include other operations on the car as well.

When using composition the front-end is in complete control of the use of the back-end. The front-end class can pick and choose which parts of the back-end behavior will be part of the front-end behavior. The front-end can combine back-end behavior with additional logic to create new front-end behavior. The front-end is under no obligation to expose all the behavior of the back-end class or to maintain behavior that is consistent with the behavior of the back-end class. Note that the front-end has no access to the internal data or methods of the back-end.

When using composition, there is no problem with a front-end class composing behavior from multiple back-end classes. The front-end picks exactly which back-end behavior is exposed and exactly how it is exposed.

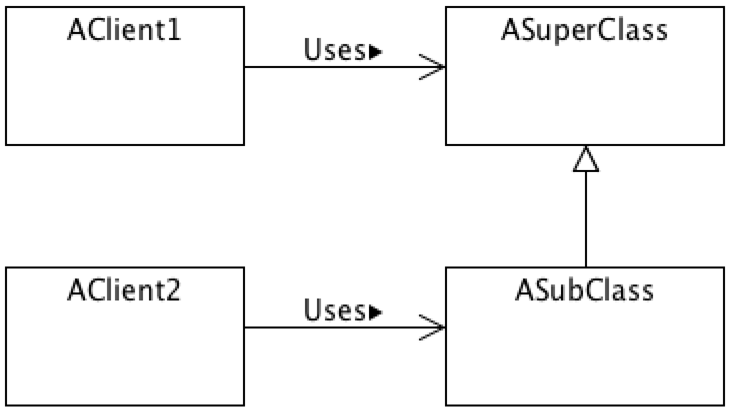
Inheritance, Composition, and Encapsulation

We want our designs to make maximum use of encapsulation or information hiding. One of the major benefits of encapsulation is to keep changes to one part of the software from “rippling” to other parts of the software.

When considering inheritance and composition to encapsulation, we need to look at how changes propagate through the software.

Change and Inheritance

For inheritance, consider the following UML diagram:



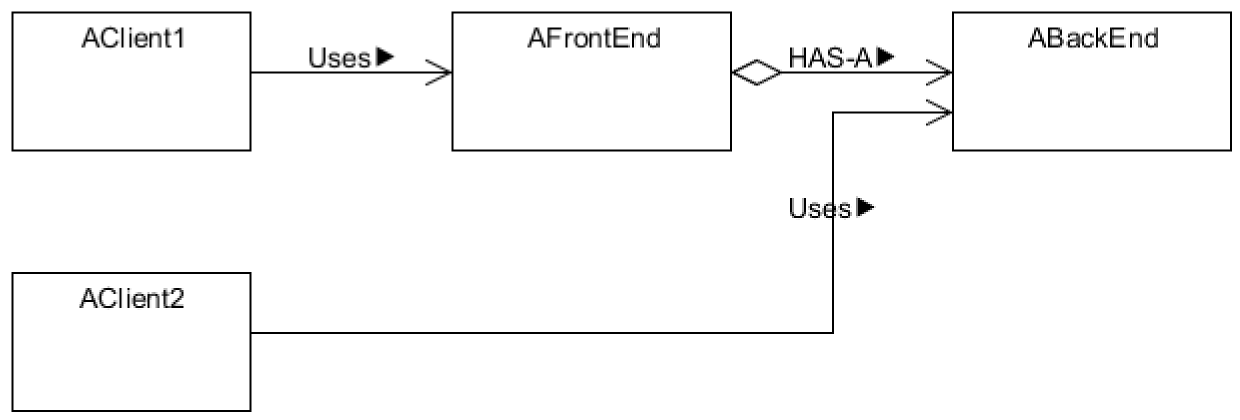
We can ask what has to change in other classes if there is a change in either the interface (externals) or implementation (internals) of ASuperClass or ASubclass (four cases total).

- If the interface of ASuperClass changes:
  - the implementation of ASuperClass has to change to match
  - the implementation of AClient1 has to be adapted for the change to ASuperClass
  - the interface of ASubclass has to change to match ASuperClass
  - the implementation of AClient2 has to be adapted for the change to ASubClass
  - the implementation of ASubclass has to change to match ASuperClass
- If the implementation of ASuperClass changes
  - the implementation of ASubclass may have to change to the extent that the subclass is sensitive to the implementation of ASuperClass
    - Note that inheritance can break encapsulation. In order to override operations in the superclass, a subclass may have to know the structure of the implementation of the superclass.
    - In Java, changes to `protected` attributes and methods of ASuperClass will not impact AClient1 but may impact ASubClass.
- You cannot change the interface of ASubclass if that would require a change to the interface of ASuperClass.
- If you change the part of the interface of ASubclass that is not part of the interface of the ASuperClass:
  - the implementation of AClient2 has to be adapted to the change.
- You can freely change the implementation of ASubclass.

One key point to note here is that the implementation of a subclass is potentially coupled to the <u>implementation</u> of its superclass. For this reason, some authors recommend that inheritance be disallowed except for classes that are designed for inheritance. See, for example, *Effective Java*, Item 17, *Design and document for inheritance or else prohibit it*.

Change and Composition

For composition, consider the following UML diagram:



We can ask what has to change in other classes if there is a change in either the interface or the implementation of AFrontEnd or ABackEnd.

- If the interface of the AFrontEnd changes:
  - the implementation of the AFrontEnd has to change to match
  - the implementation of AClient1 has to be adapted to the change
- You can freely change the implementation of AFrontEnd as long as that does not require a change to the interface of ABackEnd.
- If the interface of ABackEnd changes:
  - the implementation of AFrontEnd will have to be adapted to the change
  - the implementation of AClient2 has to be adapted to the change
- You can freely change the implementation of ABackEnd

Comparison

If you compare the two mechanisms, composition does a better job of preserving encapsulation. superclasses and subclasses are tightly coupled. Front-end and back-end classes are less tightly coupled.

When to use Composition and Inheritance

Heuristic: Liskov Substitution Principle

Instances of the subclass must be substitutable for instances of the superclass.

Another way of saying this is: *Do not do inheritance unless you have a true IS-A relationship.*

Note: In Java, inheritance is public. In C++ you can have private inheritance, that is, the subclass inherits the implementation of the superclass, but, the clients are not allowed to see that inheritance.

Examples of substitutability:

- The subclass should substitutable for a parameter to a method that expects an instance of the superclass. The following should work:

  ```
  // a method expecting Superclass as a parameter:
  ... f(Superclass x, ... ) { ... }

  // calling f with an instance of Subclass:
  Subclass y = new Subclass(...);
  f(y, ...) ;
  ```

  `f` should work correctly even though `f` is called with a parameter which is an instance of the subclass. Whatever expectations `f` has for the superclass must be preserved in the subclass.

- A method can return an instance of a subclass when the return type of the method is the superclass. The following should work:

  ```
  // a method that returns Superclass
  Superclass f(...) {
      ...
      return new Subclass(...);
  }

  // calling f with an instance of Subclass:
  Superclass y = f(...);
  ...
  ```

  The code following the call to `f` should work correctly even though `y` is an instance of a subclass of Superclass.

  In general, the subclass must support all the behavior of the superclass in a way that is consistent with the behavior of the superclass.

Example:

Consider the “List” of all prime numbers. This is a “list with infinite length”. However, in Java, this should not be considered a List. Lists have a size() method that returns the number of elements. So, a Java List cannot have an infinite number of elements. However, the list of all prime numbers is meaningful in the sense that you could write a program that iterated through the list with the list being extended if the iterator needed more elements. For example,

```
int product = 1;
for (int p : allPrimes) {
    product = product * p;
    if (product > 1000000000)
        break;
}
```

By the Liskov Substitution Principle, `allPrimes` cannot be a `List` because it does not have all the behavior expected of a `List`, and so, is not substitutable for a `List`.

Other Heuristics

Heuristic: Do not use inheritance just to get reuse or polymorphism.

You can get polymorphism by using interfaces and having multiple classes implement the interface. This allows you to have multiple classes with the same behavior without requiring that they share implementation.

You can get reuse by using composition. You are not constrained to provide consistent behavior the way you are when you use inheritance (see Liskov Substitution Principle). Example: This is a solution to the problem of wanting to make a Square a Rectangle when they don't have consistent behavior: The Square HAS-A Rectangle and delegates all the interesting work to the associated Rectangle object.

Heuristic: Keep inheritance hierarchies from getting too deep

Deep inheritance hierarchies are hard to understand. To understand an inheritance hierarchy you have to know which methods are overridden in which subclasses and which overrides call overridden methods in their superclasses.

How deep is too deep? Good question. However, something between five and ten levels is a practical limit.

Heuristic: Use composition when you are modeling "roles"

Is a Manager an Employee? Not really. Being a manager is a role that an employee can have. It's something that can come and go. It's better to model this using composition. See the example, below.

Heuristic: Use composition when you want to change behavior dynamically

Changing behavior dynamically means changing the behavior at run-time (while the program is running). You can change the composed object dynamically. For example, you can replace the Engine in a Car with a different one. This is a case where composition gives you flexibility that inheritance does not.

Heuristic: Be careful with composition if object identity is important

A back-end object does not have the same identity as the front-end object. However, an instance of a subclass has the same identity as the instance of the superclass that is "contained" within it.

Here's an example of object identity in use. Inheritance version:

```
class Displayable {
    void draw() { … }
    …
}
class Location extends Displayable {
    Displayable getDisplayable() {
        return this;
    }
    …
}
```

```
…
Location loc = … ;
List<Displayable> displayList = …;
for (Displayable d : displayList) {
    if (loc == d)
        // This code will be executed if loc is found
        // in this list.
        // loc and d can be the same object.
```

Composition version:

```
class Displayable // Same as above

class Location {
    Displayable display;
    Displayable getDisplayable() {
        return display;
    }
    …
}

…

Location loc = … ;
List<Displayable> displayList = …;
displayList.add(loc.getDisplayable());
for (Displayable d : displayList) {
    if (loc == d)
        // This code will never be executed.
        // loc and d can not be the same object.
```

Here are three possible solutions to this problem:

Alternative 1, change the test:

```
for (Displayable d : displayList) {
    if (d == loc.getDisplayable())
        // This code will be executed when d is
        // the displayable for loc.
```

This code can be brittle. It assumes that `loc.getDisplayable()` always returns the same object. (And, for example, not a new object constructed for this purpose, maybe a defensive copy.)

Alternative 2, add behavior to Location to test if a given displayable is the displayable for this Location:

```
class Location {
    Displayable display;
    Displayable getDisplayable() {
        return display;
    }
    boolean isDisplayable(Displayable d) {
        return display == d;
    }
    …
}
…
for (Displayable d : displayList) {
    if (loc.isDisplayable(d))
        // This code will be executed when d
        // is the Displayable for loc.
```

This code is preferable to the previous alternative because it makes Zone responsible for determining when d is its Displayable. (Zone is responsible for its own behavior.)

Alternative 3, make Displayable an interface and Location a Displayable:

```
interface Displayable {
    void draw();
}
class DisplayableImpl implements Displayable {
    . . . // What was Displayable
}
class Location implements Displayable {
    Displayable display;
    void draw() {
        display.draw();
    }
    …
}
…
Location loc = … ;
List<Displayable> displayList = …;
displayList.add(loc);
for (Displayable d : displayList) {
    if (loc == d)
        // This code words as expected.
```

Heuristic: Prefer composition to inheritance.

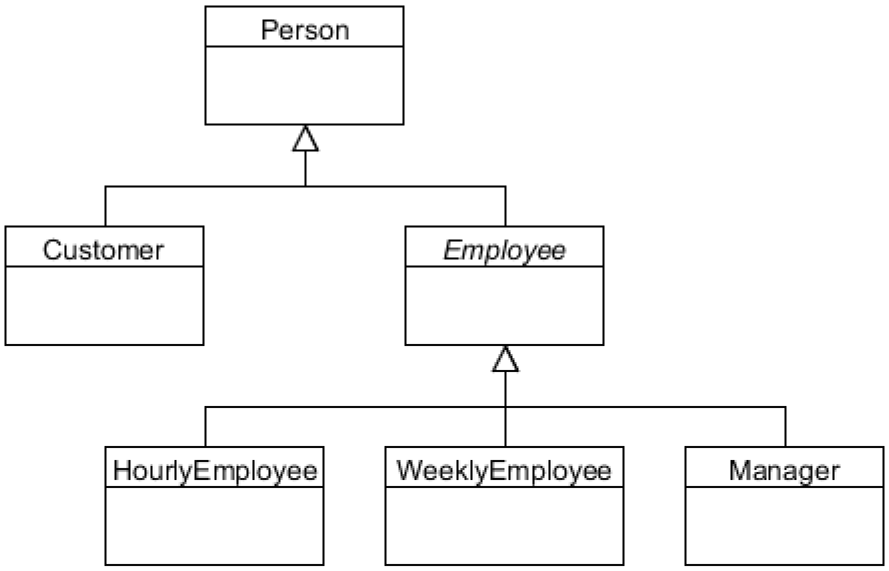
In general, if you aren't sure whether you want to use inheritance or not, use composition.

An Example

Consider using inheritance:

```
class Person { }
class Employee extends Person { }
class HourlyEmployee extends Employee { }
class WeeklyEmployee extends Employee { }
class Customer extends Person { }
class Manager extends Employee { }
```

Here's the picture:



What's wrong here? How about:

- A Person can not be both a Customer and an Employee.
- Is a Manager an HourlyEmployee or a WeeklyEmployee or …

A better idea using composition:

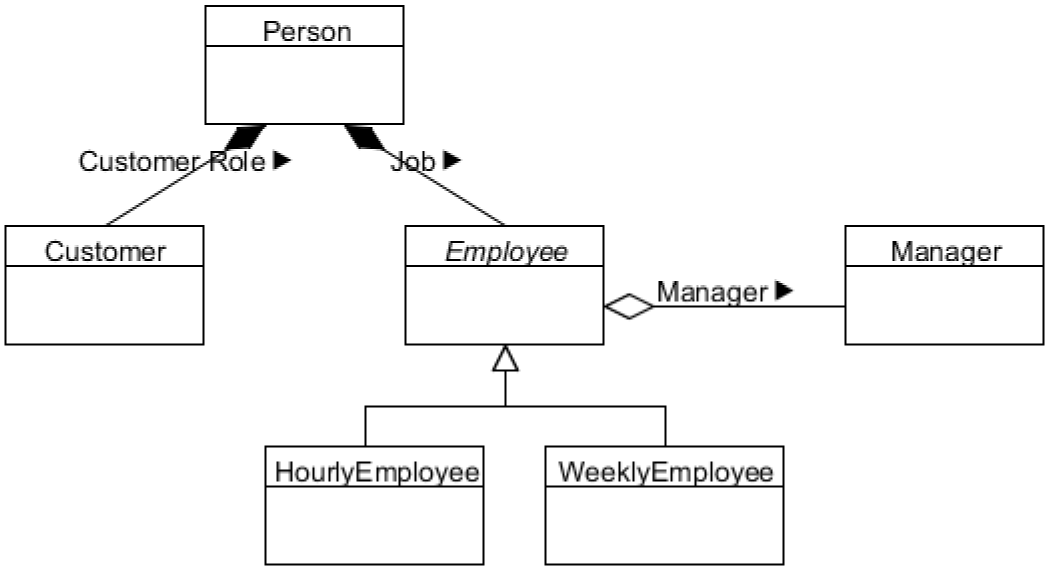
```
class Person { }
class Employee { }
class HourlyEmployee extends Employee { }
class WeeklyEmployee extends Employee { }
class Customer { }
class Manager { }
```

Where:

- Employee and Customer are roles for a Person. This allows Persons to dynamically change roles, to have more than one job, to be both a customer and an employee, etc.
- HourlyEmployee and WeeklyEmployee are still specializations of Employee.

- Manager is a role for an Employee.

Here's the new picture:


Person
Customer Role ▶
Job ▶
Customer
Employee
Manager ▶
Manager
HourlyEmployee
WeeklyEmployee