Inheritance

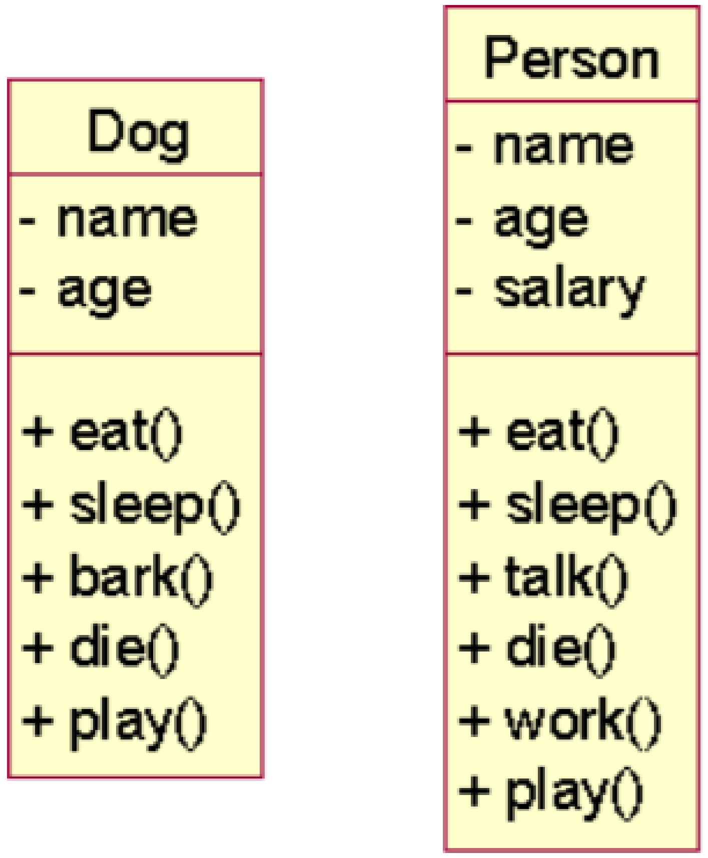
One of the most well-known and most powerful concepts behind Object Orientation is **Inheritance**. In this chapter, we'll have a basic look at inheritance, when to apply it (and when not to apply it), and in particular we'll look at the UML notation for expressing inheritance.

Lecture 9

Inheritance – the basics

Often, several classes in a design will share similar characteristics. In Object Orientation, we can factor out these common characteristics into a single class. We can then “inherit” from this single class, and build new classes from it. When we have inherited from a class, we are free to add new methods and attributes as the need arises.

Here's an example. Let's say we are modelling the attributes and behaviour of Dogs and People. This is what our classes might look like:



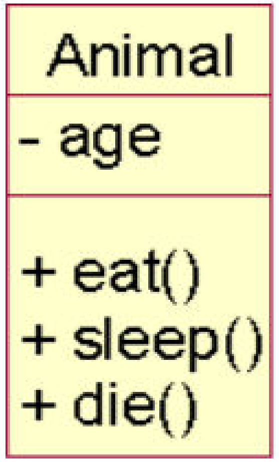
Modelling Dogs and People

Although the two classes are different, the two classes also share a lot in common. Every Person has a name; so too does every Dog[14]. Similarly with Age. Some of the behaviours are common, such as Eat, Sleep and Die. Talk, however, is unique to the Person class.

If we decided to add a new class to our design, such as "Parrot", it would be tedious to add all of the common attributes and methods to the Parrot class again. Instead, we can factor out all of the common behaviour and properties into a new class.

[14] Let's assume we are modelling *pet* dogs!

If we take out the attribute "Age", and the behaviours "Eat", "Sleep" and "Die", we have attributes and behaviour which should be common amongst *all* animals. Therefore, we can build a new class called "Animal".



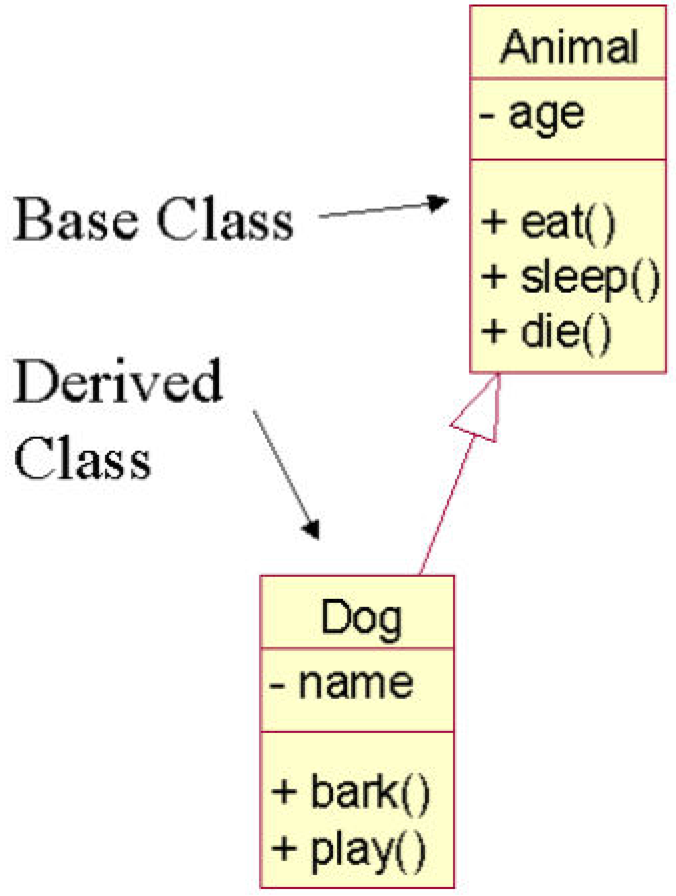
A more general "Animal" class

So we have now built a more general class than our specific Dog, Person and Parrot class. This process is known as **generalisation**.

Now, when we need to create the Dog class, instead of starting from scratch, we inherit from the Animal class and merely add on the attributes and methods that we need for our specific class.

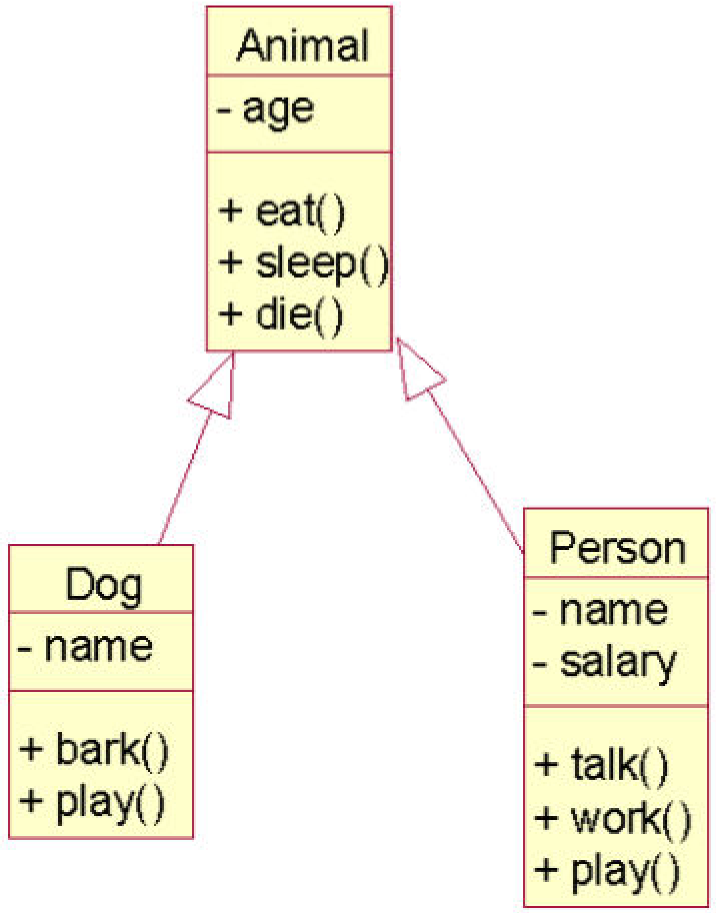
The following diagram illustrates this in the UML. Note that in the new Dog class we don’t include the old methods and attributes – they are implicit.

The class we started from is called the **base class** (sometimes called the Superclass). The class we have created from it is called the **derived class** (sometime the Subclass).



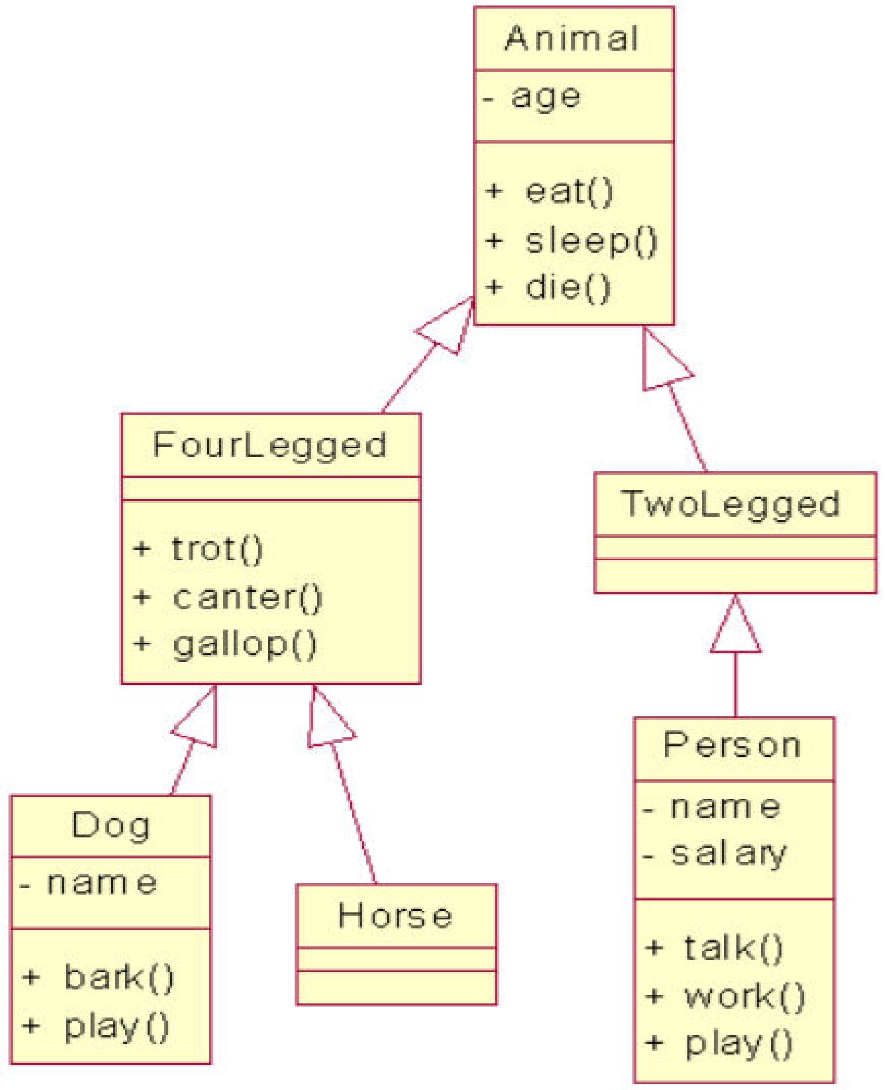
Creating a Dog class from the Animal Class

We can continue creating new derived classes from the same base class. To create our Person class, we inherit again, as follows:



Person derived from Animal

We can continue inheriting from classes to form a *class hierarchy*.



A Class Hierarchy

Inheritance is White Box Reuse

A common mistake in object oriented designs is to over use inheritance. This leads to maintenance problems. Effectively, a derived class is tightly coupled to the base class – changes to the base class will result in changes to the derived class.

Also, when we use a derived class, we need to find out exactly what the base classes can do. This might mean trawling through a large hierarchy structure.
This problem is known as the *proliferation of classes*.

One common cause of proliferation is when inheritance is used when it shouldn't be.
Follow the following rule-of-thumb:

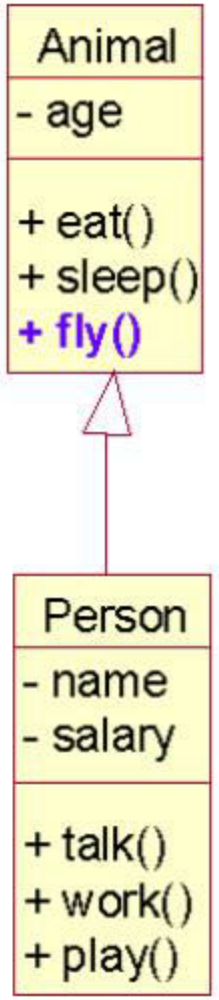
Inheritance should only be used as a generalisation mechanism.

In other words, only use inheritance when the derived classes are a specialised type of the base class. There are two rules to help here:

- The **is-a-kind-of** rule
- The **100%** rule

The 100% Rule

All of the base class definition should apply to all of the derived classes. If this rule doesn't apply, then when you inherit, you are not creating specialised versions of the base class. Here's an example:



Poor inheritance

the fly() method should not be part of the Animal class. Not all animals can fly, so the derived class, Person, has an extraneous method associated with it.

Ignoring the 100% rule is an easy way to create maintenance problems.

Substitutability

In the previous example, why could we not simply **remove** the "fly" operation in the person class? That would solve the problem.

Methods **cannot** be removed in an inheritance hierarchy. This rule is enforced to ensure that the **Substitutability Principle** is upheld. We'll look at this in a little more detail shortly.

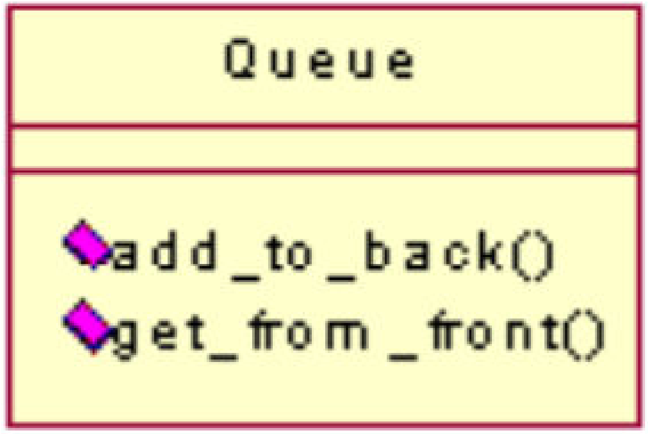
The Is-A-Kind-Of Rule

The Is-A-Kind-of rule is a simple way to test if your inheritance hierarchy is valid. The phrase "<derived class> is a <base class>" should make sense. For example "a dog is a kind of animal" makes sense.

Often, classes are derived from base classes when this rule does not apply, and again, maintenance problems are likely. Here's a worked example:

Example - Reusing queues through inheritance

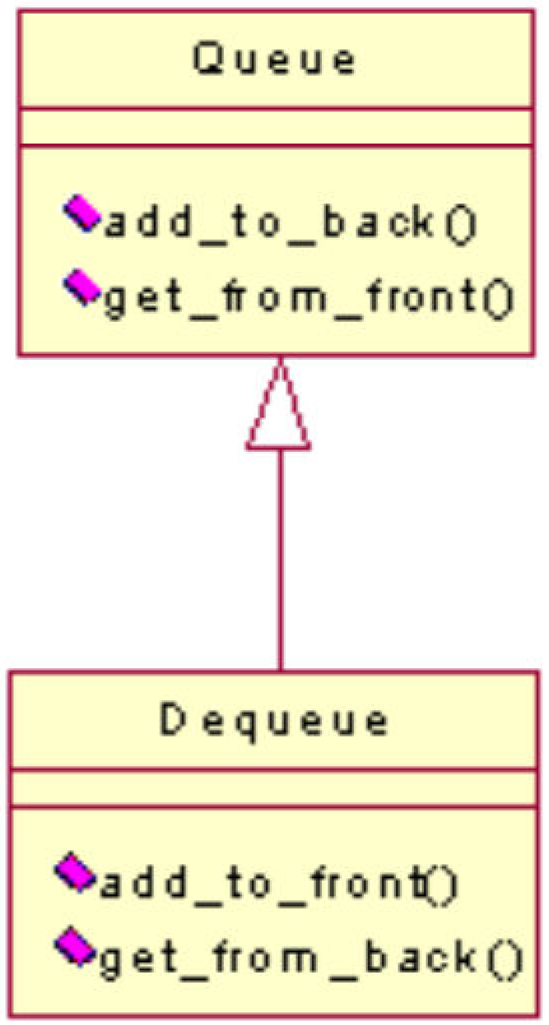
Assume that we have built (in code) a working Queue class. The Queue class allows us to add items to the back of the queue and remove items from the front of the queue.



The Queue Class

After a while, we decide that we need to build a new type of queue – a special kind of queue called a "Dequeue". This kind of queue allows the same operations as a queue but with the additional behaviour of allowing items to be added to the front of the queue and for items to be removed from the back. A kind of "two-way" queue.

To reuse the work we have already completed with the Queue, we can inherit from the Queue class.



Building a Dequeue through inheritance

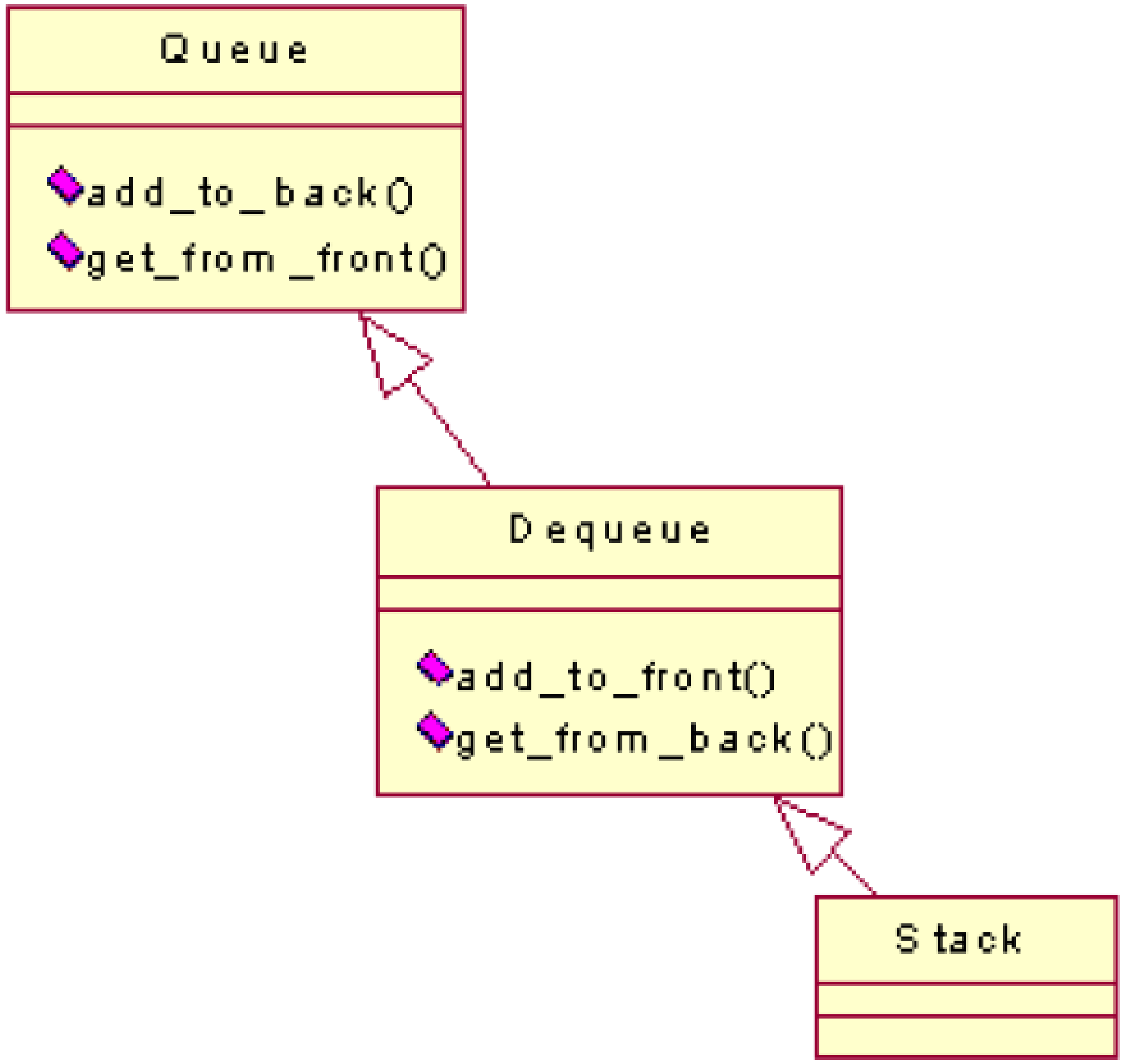
Does this pass the Is-A-Kind-of and 100% tests?

- **100%** : Do all of the methods in Queue apply to Dequeue? The answer is yes, because all of these methods are required.
- **Is-A-Kind-Of** : Does this sentence make sense? "A Dequeue is-a-kind-of Queue". Yes, it does, because a dequeue is a special kind of queue.

So this inheritance was **valid**.

Now, let's go further. Let's say that we need to create a Stack. For a stack, we need to support the methods add_to_front() and remove_from_front().

Rather than writing the stack from scratch, we could simply inherit from the dequeue, as the dequeue provides both of these methods.



Creating a Stack...no work needed!

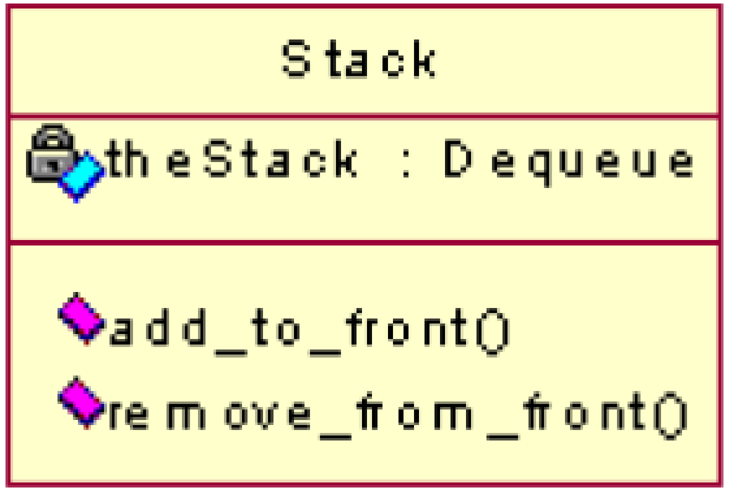
We can feel very smug that we have re-used the Dequeue and created a stack with no further work required. Any code using the stack can add items to the front, and remove them from the front, and therefore we have a working stack.

We shouldn't feel too smug though. *This is a very poor design indeed*. Don't forget that the stack has also inherited the methods add_to_back() and remove_from_back() – these are two methods that are meaningless in a stack! So the stack fails the **100%** test. In addition, the stack fails the **Is-A-Kind-Of** test, because a stack isn't really a kind of Dequeue at all!

So we've created a maintenance problem – namely that any code using the stack can erroneously add items to the back of the stack! How do we get around this?? There really isn't any way, other than to bodge the stack class. Remember that we cannot remove methods from a class when we inherit. (The **substitutability** principle)

The solution is to use aggregation rather than inheritance. We create a new class called **stack**, and include a Dequeue as one of its private attributes.

We can now provide the two public methods, add_to_front() and remove_from_front() as part of the stack class. The implementation of these methods are simple calls to the same methods contained in the Dequeue.



The Stack class reusing the Dequeue efficiently

Now, users of the Stack class can only call the two public methods – the methods contained within the “hidden” Dequeue are private. This ensures that the Stack class has a highly cohesive interface, and will be much easier to maintain and understand.

Problems With Inheritance

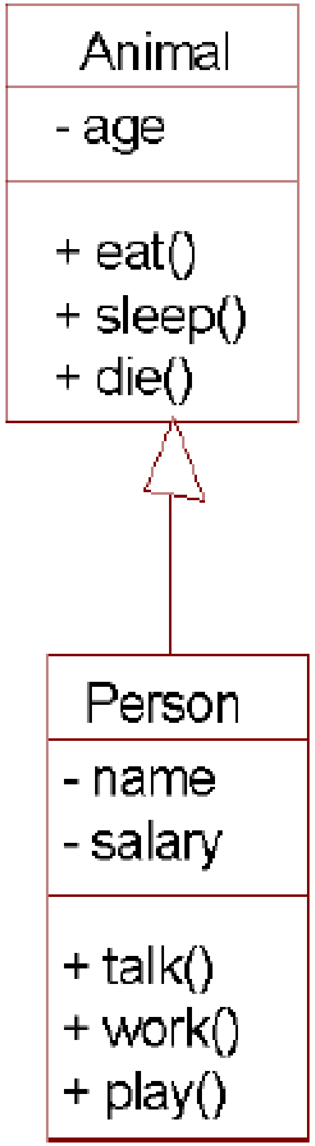
Although Inheritance looks like a powerful mechanism to achieve reuse, Inheritance should be approached with care. Overusing inheritance can lead to very complex and difficult to understand hierarchies. This problem is known as the *proliferation of classes*. The problem is made even more acute when inheritance is used incorrectly (as described above). So ensure that inheritance is used sparingly, and make sure the 100% and is-a-kind-of rules apply.

In addition, Inheritance is White Box Reuse. Encapsulation between the base class and the derived class is quite weak - generally, a change to the base class could impact any derived classes, and certainly, any user of a class also needs to know about how the chain of parent classes above the class works.

The user of a class that is buried at the foot of an 13-class deep inheritance chain is going to have a real headache when working with that class - how many classes can *you* juggle in your head at once??

Visibility of Attributes

Consider the inheritance tree



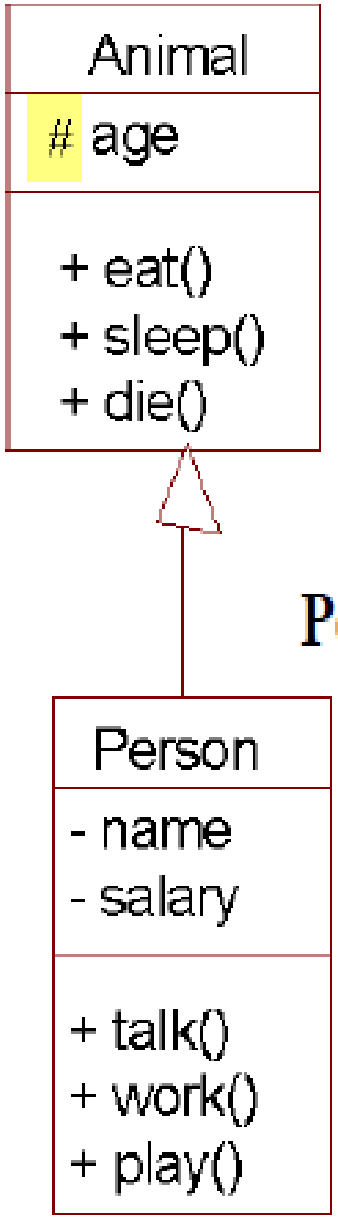
It is important to realise that the private members of the base class, Animal are **not** visible to the derived class, Person. So the methods talk(), work() and play() cannot access the age attribute.

This makes sense in a way, because you can argue that the methods that are only relevant to the Person class should not be able to fiddle with the attributes of the Animal class.

However, this restriction is sometimes too tight, and you need to allow a derived class to be able to "see" the attributes in the base class. Of course, we could make the attributes public, but that would break encapsulation and open up the attributes to the entire world. So OO provides a "middle ground", called **protected visibility**.

A protected member is still private to the outside world, but will remain visible to any derived classes. Most OO languages support protected visibility.

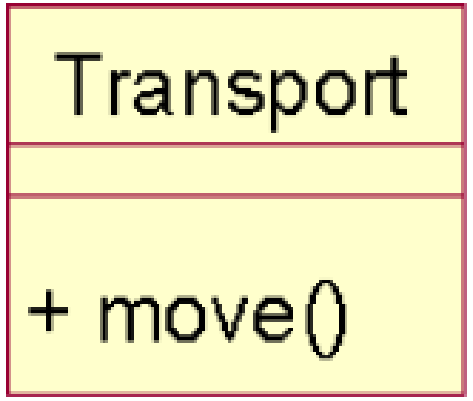
The UML notation for a protected member is shown below:



The Age attribute is now protected, and is therefore visible to the Person class. It is still "private" as far as other classes are concerned.

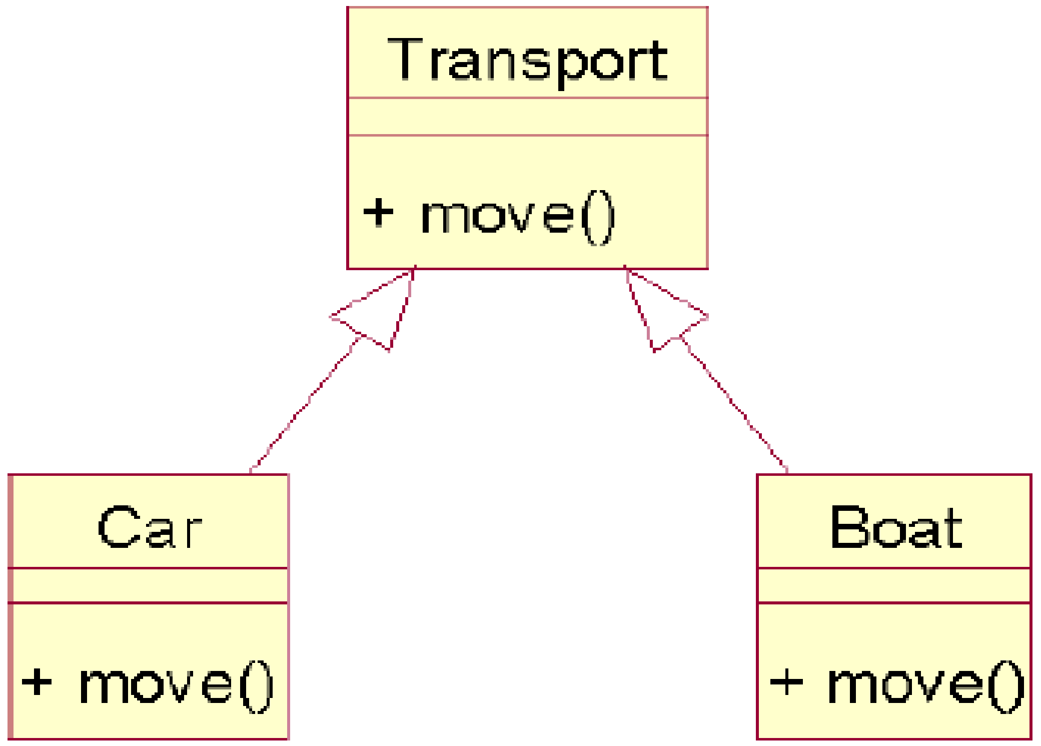
Polymorphism

Derived classes can redefine the implementation of a method. For example, consider a class called “Transport”. One method contained in transport must be move(), because all transport must be able to move:



The Transport Class

If we wanted to create a Boat and a Car class, we would certainly want to inherit from the Transport class, as all Boats can move, and all Cars can move:



Boat and Car derived from Transport. The Is-A-Kind-Of and 100% rules are satisfied.

However, cars and boats move in different ways. So we will probably want to implement the two methods in different ways. This is perfectly valid in Object Orientation, and is called **Polymorphism**.

Abstract Classes

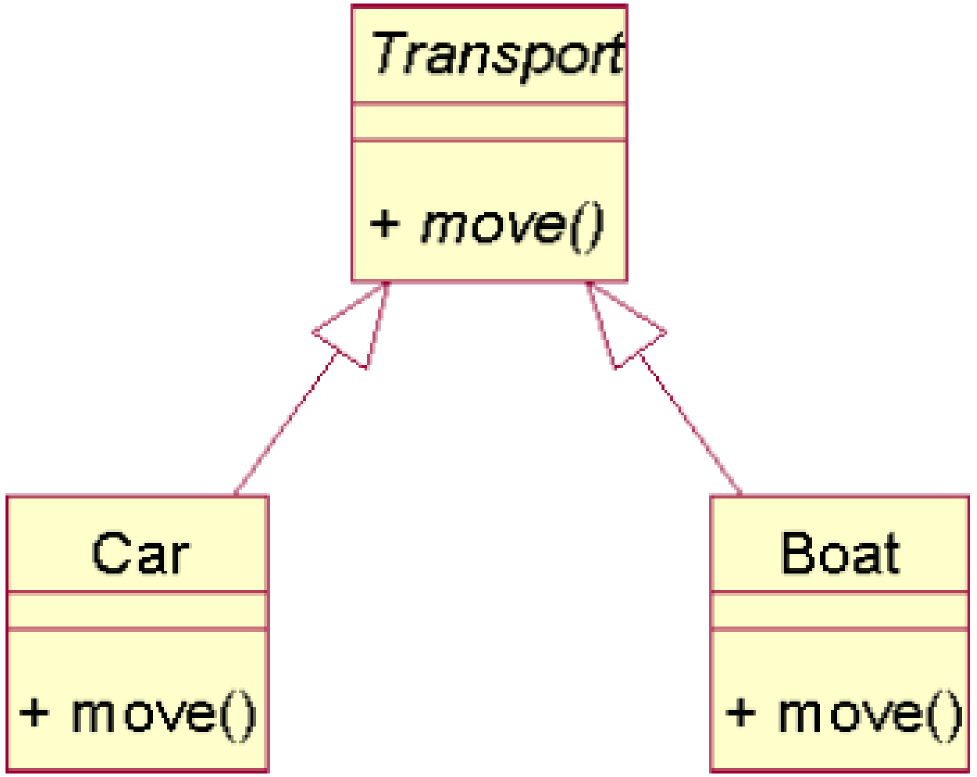
Often in a design, we need to leave a method unimplemented, and defer its implementation further “down” the inheritance tree.

For example, back to the situation above. We added a method called “move()” to Transport. This is good design, because all transport needs to move. However, we cannot really implement this method, because Transport is describing a wide range of classes, each with different ways of moving.

What we can do is make the Transport class **abstract**. This means that some, or maybe all, of the methods are unimplemented.

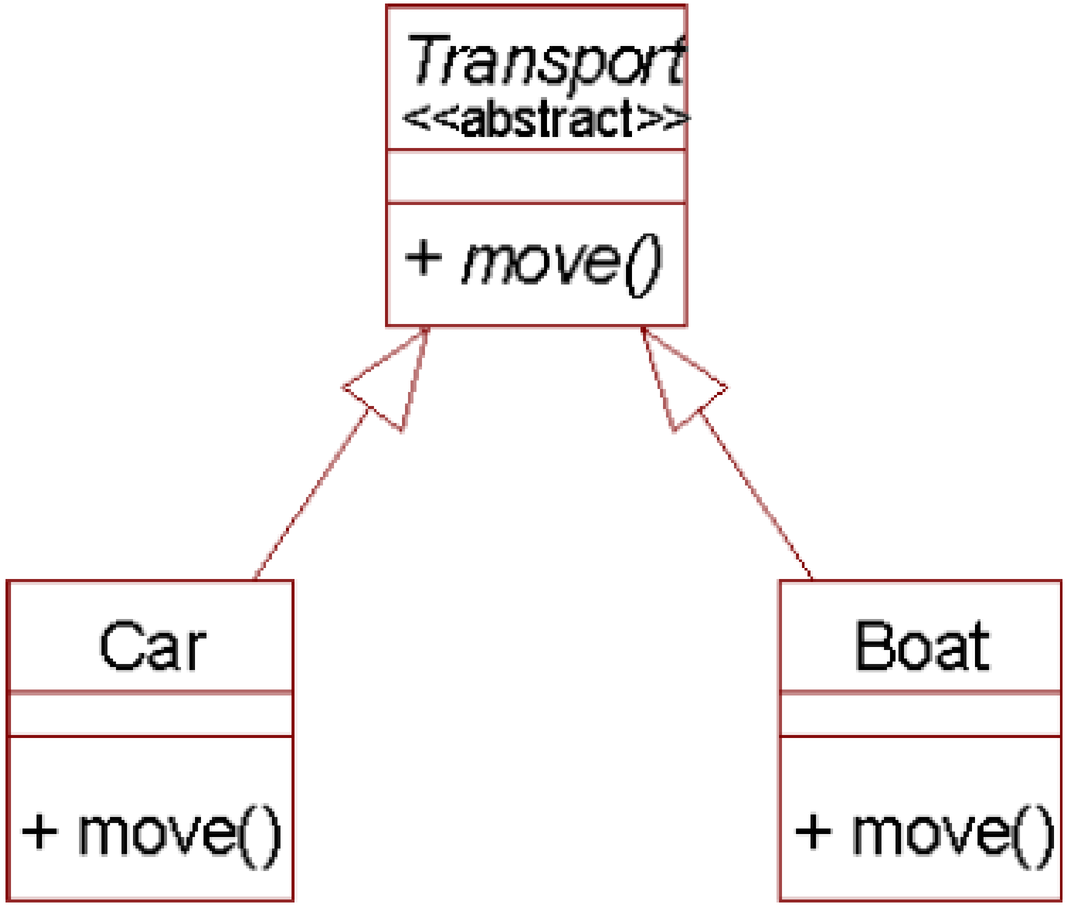
When we derive the car class from Transport, we can then go ahead and implement the method, and similarly in boat.

The UML Syntax for an abstract class and an abstract method is to use italics, as follows:



We don't implement the move() method in Transport

This is one area of the UML that I don't think has been too well thought out. Italics are often difficult to spot on a diagram (and difficult to produce if you are writing the diagram on paper). The solution is to use a UML Stereotype on the abstract class as follows:



Clarifying an Abstract class using a stereotype

The Power of Polymorphism

Polymorphism comes into its own when we apply the principle of substitutability. This principle says that any method we write that expects to "work on" a particular class, **can happily work on any derived class too**.

```
public void accelerate (Transport theTransport, int
acceleration)
{
    -- some code here
    theTransport.move();
    -- some more code
}

--
--
--
accelerate (myVauxhall);
accelerate (myHullFerry);
```

Java code illustrating substituability

In this example, I have written a method called accelerate. It works using a parameter of type "Transport", and presumably speeds up the transport using the move() method.

Now, I can safely call this method, and pass it a Car object (because it is a subclass of Transport), and I can safely pass it a Boat object too. The function I have written simply doesn't care what the actual type of the object is, as long as it is derived from Transport.

This is extremely flexible. Not only have I written a general purpose method that can work on a whole range of different classes, I have also written a method that could in future be used on a class that isn't even designed yet. Later, someone might create a new class called "Aeroplane", derived from Transport, and the accelerate() method would still work, and happily accept the new Aeroplane class, **without modification or recompilation!**

This is the reason why we cannot remove a method when we derive a new class. If we were allowed to do so, the aeroplane class could conceivably remove the move() method, and all the benefits listed above would be destroyed!

Summary

The Notation for Inheritance in UML is simple.

Classes can be arranged into an “inheritance hierarchy”

A sub-class *must* inherit all of the parent class’ public behaviour

Protected methods and attributes are also inherited

Polymorphism is an incredibly powerful tool to achieve code reuse

Design Patterns

GangOfFour

Defining design patterns

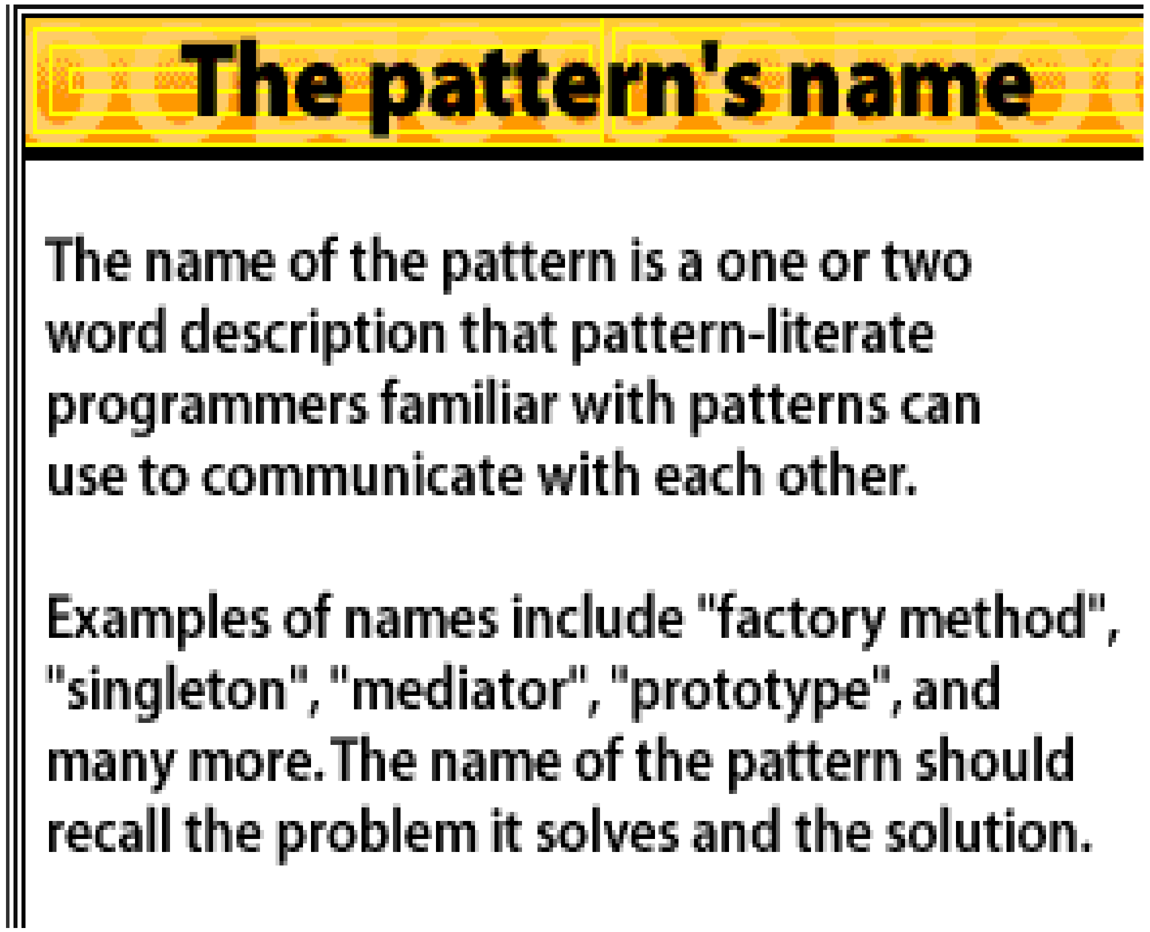
- If a problem occurs over and over again, a solution to that problem has been used effectively. That solution is described as a **pattern**.
- The design patterns are language-independent strategies for solving common object-oriented design problems.
- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern is a description or template for how to solve a problem that can be used in many different situations.

Defining design patterns

- Code reuse is one of the prime benefits of object-oriented programming.
- Rather than solving a problem from scratch, you simply derive a *subclass* from an existing class and override a few methods.
- The most common purpose of design patterns is to make classes reusable and extensible.
- Design patterns do this by documenting common solutions to common problems.

Defining design patterns

- Each design pattern has four essential elements:

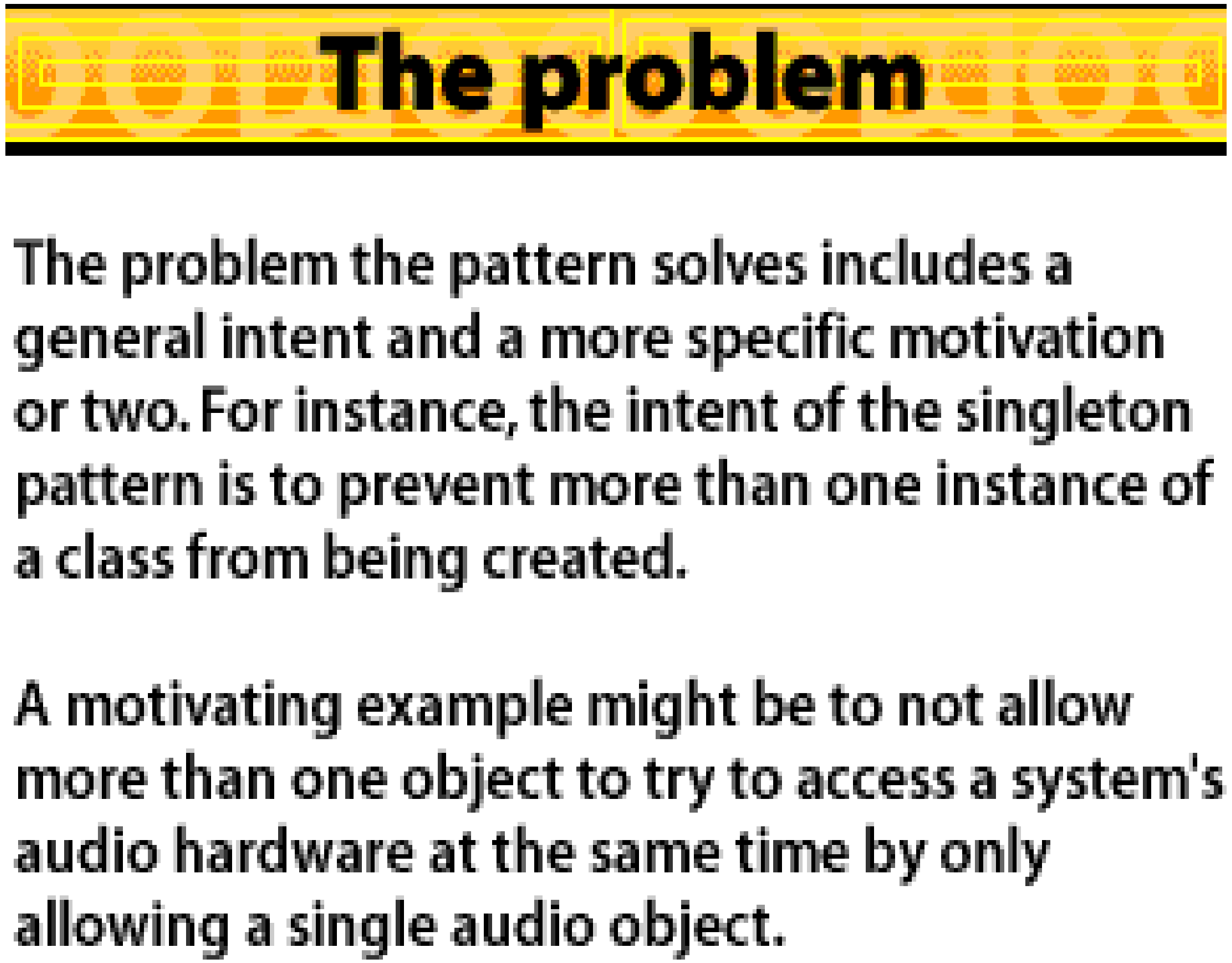


The pattern's name

The name of the pattern is a one or two word description that pattern-literate programmers familiar with patterns can use to communicate with each other.

Examples of names include "factory method", "singleton", "mediator", "prototype", and many more. The name of the pattern should recall the problem it solves and the solution.

Defining design patterns

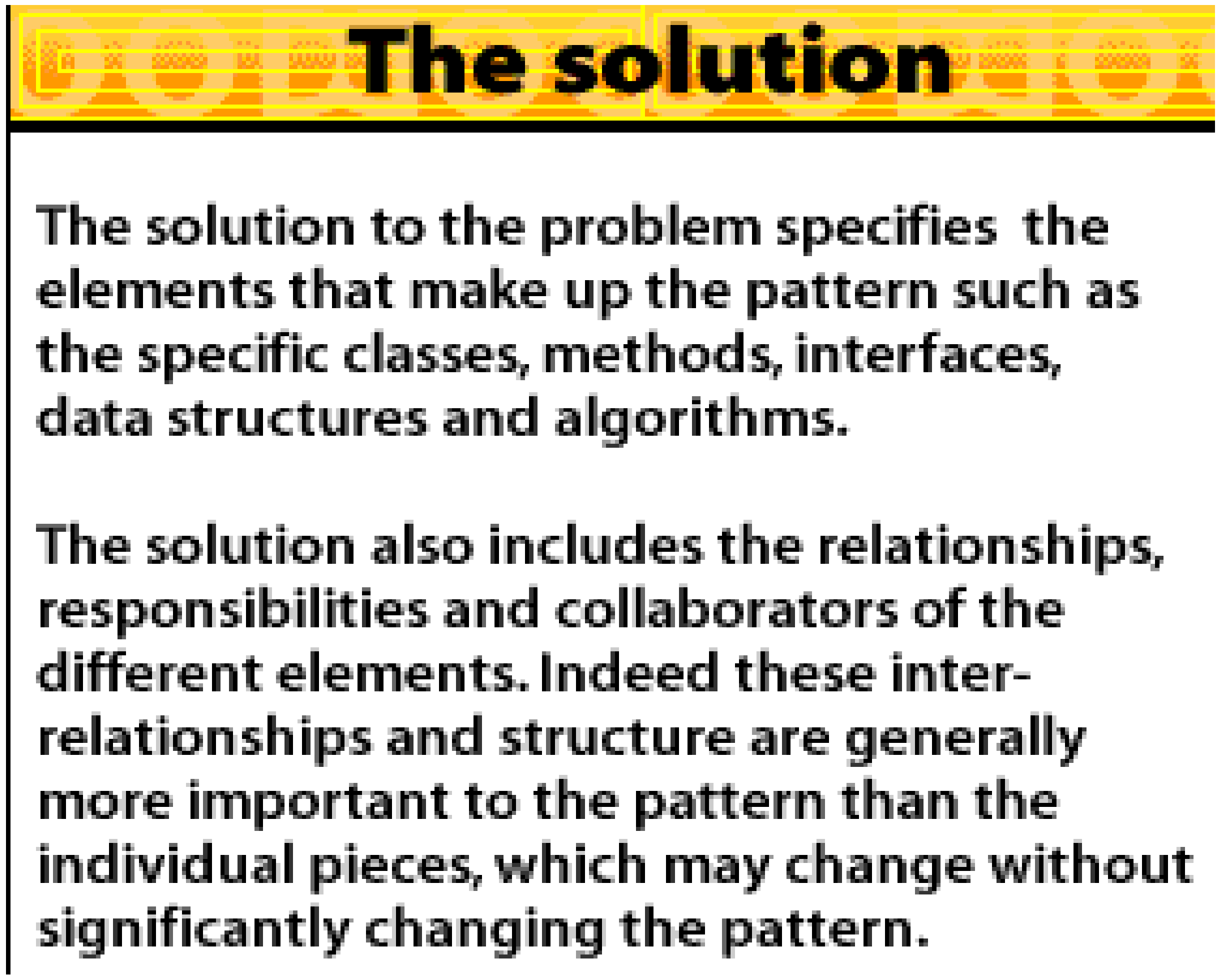


The problem

The problem the pattern solves includes a general intent and a more specific motivation or two. For instance, the intent of the singleton pattern is to prevent more than one instance of a class from being created.

A motivating example might be to not allow more than one object to try to access a system's audio hardware at the same time by only allowing a single audio object.

Defining design patterns

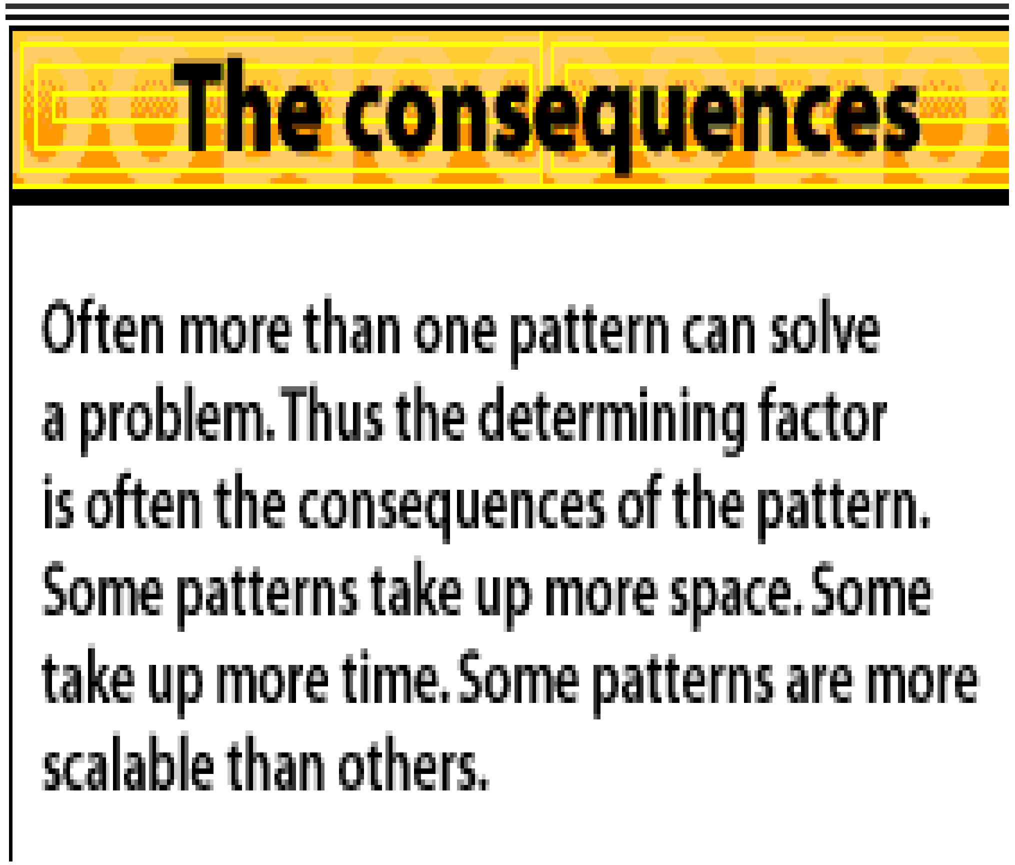


The solution

The solution to the problem specifies the elements that make up the pattern such as the specific classes, methods, interfaces, data structures and algorithms.

The solution also includes the relationships, responsibilities and collaborators of the different elements. Indeed these inter-relationships and structure are generally more important to the pattern than the individual pieces, which may change without significantly changing the pattern.

Defining design patterns



The consequences

Often more than one pattern can solve a problem. Thus the determining factor is often the consequences of the pattern. Some patterns take up more space. Some take up more time. Some patterns are more scalable than others.

Why use design patterns?

- Design patterns let you write better code more quickly. Of the five phases of software development, design patterns do almost nothing in the analysis, testing, or documentation phases.
- Design patterns, as the name implies, have their biggest impact in the design phase of a project.

1. Design patterns help you to write code

Delegation: This pattern is one where a given object provides an interface to a set of operations. However, the actual work for those operations is performed by one or more other objects.

Composition: Creating objects with other objects as members. Composition should be used when a *has-a* relationship applies.

Why use design patterns?

2. Design patterns encourage code reuse and accommodate change by supplying well-tested mechanisms for *delegation* and *composition*, and other non-inheritance based reuse techniques.

3. Design patterns encourage more maintainable code

4. Design patterns provide a common language for programmers.

Design patterns are divided into three fundamental groups:

- **Creational patterns**
- Creational patterns are used to create objects of the right class for a problem, generally when instances of several different classes are available.
- They're particularly useful when you are taking advantage of polymorphism and need to choose between different classes at runtime rather than compile time.

Design patterns are divided into three fundamental groups:

- **Structural patterns**
- Structural patterns form larger structures from individual parts, generally of different classes.
- Structural patterns vary a great deal depending on what sort of structure is being created for what purpose.

Design patterns are divided into three fundamental groups:

- **Behavioral patterns**
- Behavioral patterns describe interactions between objects.
- They focus on how objects communicate with each other.
- They can reduce complex flow charts by limiting interconnections between objects of various classes.
- Behavioral patterns are also used to make the algorithm that a class uses simply another parameter that's adjustable at runtime.

There are two pattern scopes:

- Patterns may be further divided according to their scope. **Object patterns**
- Object patterns, the more common of the two, specify the relationships between objects.
- In general, the purpose of an object pattern is to allow the *instances* of different classes to be used in the same place in a pattern.
- Object patterns mostly use *object composition* to establish relationships between objects.

There are two pattern scopes:

Class patterns

- Class patterns specify the relationship between classes and their subclasses.
- Thus, class patterns tend to use inheritance to establish relationships.

Class scope is defined at design time and is built in the structure and relationship of classes where as object scope is defined at runtime and is based on the relationship of objects.

Next week Examples for Design Patterns & System Architecture…