7SENG003W Advanced Software Design

# Generalization, Inheritance and Polymorphism

Simon Courtenage
University of Westminster

# Welcome back

- Updated schedule – starting from Week 7

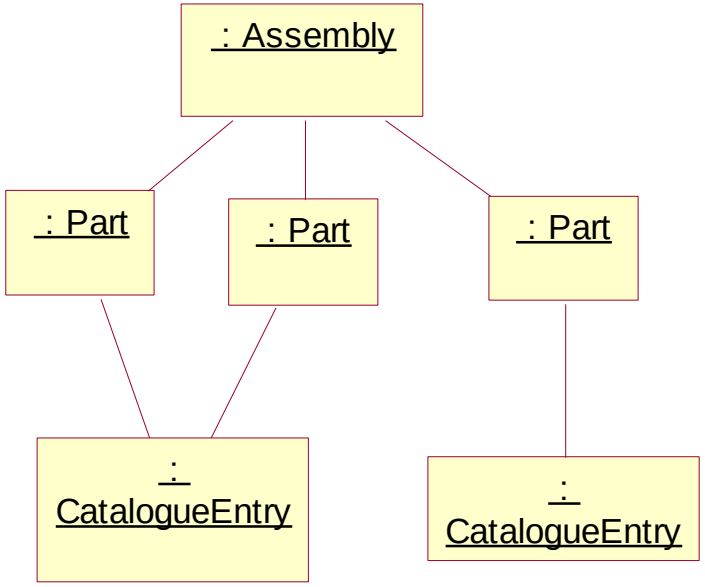
7. Inheritance and polymorphism
8. Inheritance and polymorphism (contd.)
9. Design patterns and Design Heuristics – refining designs
10. Design Patterns (contd.)
11. Statecharts
12. Testing frameworks / summary

# This week

- Start thinking about why we need generalization – using Inventory example
  - Look at the kind of problem that generalization helps with
- How to represent generalization in UML (and in Java)
- Relationship between generalization, inheritance, and polymorphism
  - What these terms mean

# Why we might need generalisation…

- In the stock control example, parts are used to make assemblies
- Information about the parts used to make the assembly is held implicitly – the assembly object has links to the relevant part objects
- An implementation of the assembly class will need a data structure to store all the links to part objects, and some way to add/delete links from this data structure

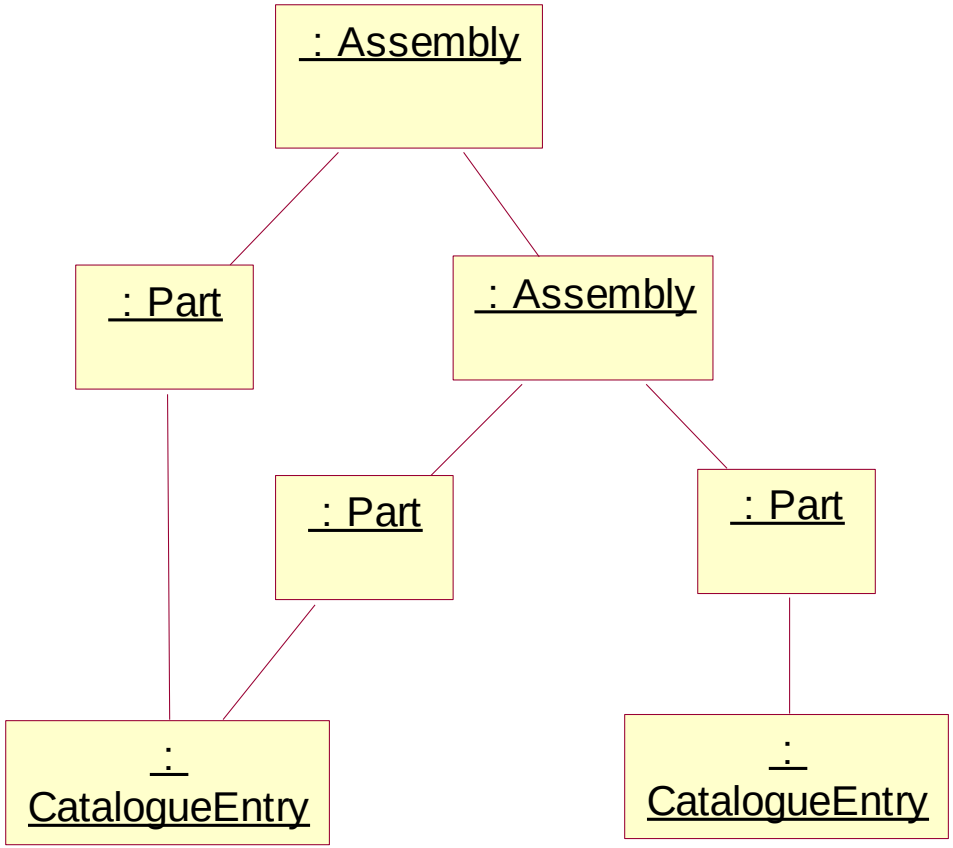


# A Simple Implementation

```
6   import java.util.*;
7
8
9   public class Assembly {
        private List<Part> components;
11
12      public Assembly() {
            components = new ArrayList<Part>();
14      }
15      public void add(Part p) {
16          components.add( e:p);
17      }
18      public int size() {
19          return components.size();
20      }
21  }
```

# Complex Assemblies

- What if we wanted to build more complex assemblies, part of whose substructure was not a simple part, such as a screw or a bolt, but another assembly?



# Problems

- How should we store the links to sub-assemblies?
  - C# and Java (as well as UML) are strongly-typed languages, adding Assembly objects to the collection of Part objects would be illegal
  - We could maintain two separate lists, one for Parts and another for Assemblies
  - **But - duplication of code plus coding problems**
  - **how, for example, should you find out the cost of an assembly?**

# New Assembly code for sub-assemblies

```
6  import java.util.*;
7
8
9  public class Assembly {
       private List<Part> components;
11     private List<Assembly> assemblies;
12
13     public Assembly() {
           components = new ArrayList<Part>();
15     }
16     public void add(Part p) {
17         components.add( e:p);
18     }
19     public void add(Assembly a) {
20         assemblies.add( e:a);
21     }
22     public int size() {
23         int assembly_sz = 0;
24         for (Assembly a : assemblies) {
25             assembly_sz = assembly_sz + a.size();
26         }
27         return components.size() + assembly_sz;
28     }
29 }
```

Note we have to duplicate the add() method

# Using the new Assembly class

```
11  public class Inventory {
12
13      public static void main(String[] args) {
14          // create objects using 'new' operator and appropriate constructor
15          CatalogueEntry screws = new CatalogueEntry( nm:"screw", num:1234, c:0.02);
16          CatalogueEntry nails = new CatalogueEntry( nm:"nail", num:1235, c:0.015);
17
18          Part screw1 = new Part( c:screws);
19          Part screw2 = new Part( c:screws);
20          Part nail1 = new Part( c:nails);
21
22          System.out.println(screw1.getName() +
23                  " costs " + screw1.getCost());
24
25          Assembly a1 = new Assembly();
26          Assembly a2 = new Assembly();
27          a1.add( p:screw1);
28          a2.add( p:screw2);
29          a2.add( p:nail1);
30          a1.add( a:a2);
31
32          System.out.println("Assembly a1 has " + a1.size() + " parts");
33
34
35      }
36  }
```

# Parts and assemblies

- Within an Assembly object, we want to do **the same thing** to other Assembly objects (sub-assemblies) that we do to Part objects
  - Add to Assembly
  - Get cost
- In terms of requests we make of them, we want to treat Parts and Assemblies as **being the same**
  - Even though they are different classes/types
- Dilemma: we want to use the same code on values of different classes but strong typing means (ordinarily) we can't – how do we solve this dilemma

# Polymorphism

- A solution to this dilemma would be to treat parts and assemblies as interchangeable objects
  - I.e, that a variable could hold either a Part object or an Assembly object
  - A Part object can be substituted for an Assembly object and vice-versa
- Known as polymorphism, which simply means “many types”. It means that where we expect a value of one type, we can use a value of another type – substitution
- How can we get/use polymorphism but respect strong typing?

# Strong typing

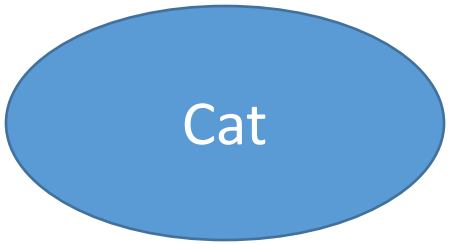
- A type (esp in OOP) is a kind of **contract**
  - A value with a particular type will fulfill all requests/behaviour expected as a result of the type

```
int i;

String s;

Person p;
```

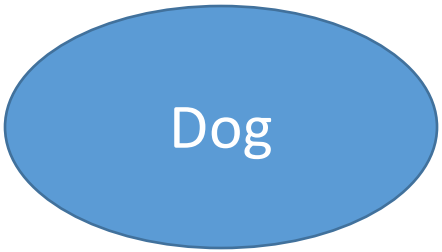
The values stored in these variables will behave as expected given their types

  - Hence a compiler can check that what we do to a value matches the contract represented by its type and report any errors
- Polymorphism?

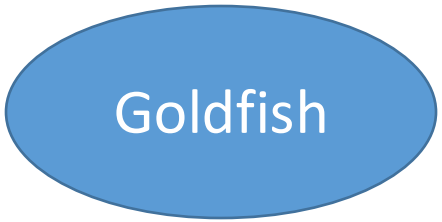
# How could we treat these things as the same?




Needs feeding




Needs feeding



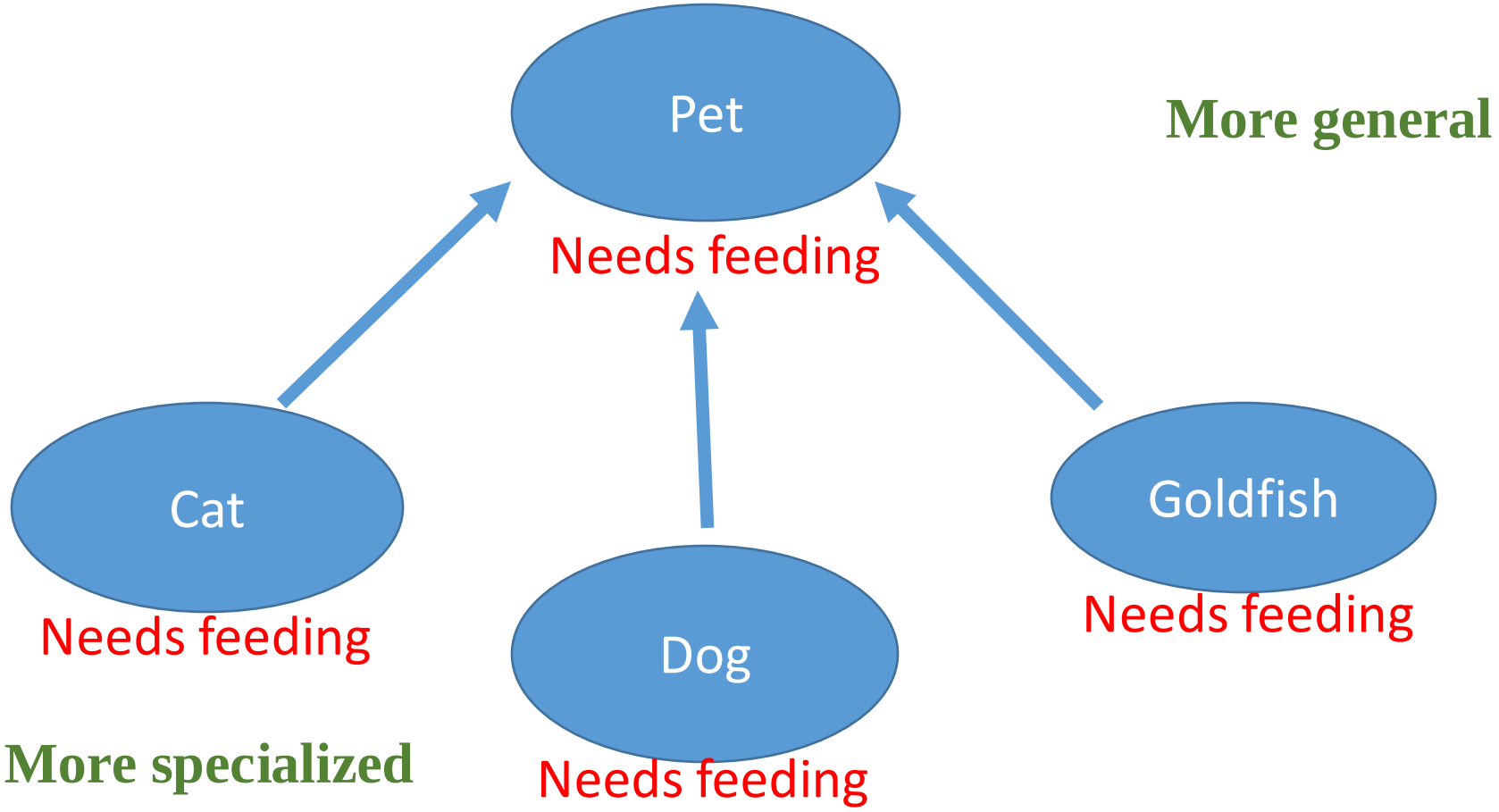

Needs feeding

# Creating and using polymorphism

- We create polymorphism by
  1. defining a new class which is more general version of the different classes whose objects we want to use polymorphically/interchangably
     - For example, if we want to make Assembly and Part objects interchangeable, we create a class which is a more general version of both Assembly and Part
  2. Stating that each of the classes is a specialized version (or subclass) of the new general class (superclass)
  3. Wherever we want to use values of the subclasses, we specify the superclass

- We'll show this (i) in terms of code, and then (ii) in terms of UML

# Dogs, cats, goldfish are kinds of Pet



# Code for Polymorphism

1. The generalized form (or base class or superclass) of both Assembly and Part

```
public class Component {
...
}
```

2. Make Assembly and Part **specialized versions** (or derived/specialised class or subclasses) of Component

```
public class Assembly extends Component {
...
}

public class Part extends Component {
...
}
```

# Component class

```
6
    abstract public class Component {
        abstract public double getCost();
9       public int size() { return 0; }
10  }
```

Note: describing the class as “abstract” means that we don’t want to actually create objects from this class.

We’ll use this class as the “base” class for creating more specialized classes – any methods declared here will also form part of the specialized classes.

# Part subclass of Component

```
3    // Part now uses Component as its base class
4    public class Part extends Component {
5        private CatalogueEntry entry;
6
7        Part(CatalogueEntry c) {
8            entry = c;
9        }
10       Part(Part p) {
11           entry = p.entry;
12       }
13
14       public String getName() {
15           return entry.getName();
16       }
17       public int getNumber() {
18           return entry.getNumber();
19       }
20
21       @Override
22       public double getCost() { // override abstract method from base class
23           return entry.getCost();
24       }
25       @Override
26       public int size() { return 1; }
27   }
```

# Assembly subclass of Component

```
9   public class Assembly extends Component {
        private List<Component> components;
11
12      public Assembly() {
            components = new ArrayList<Component>();
14      }
15      public void add(Component p) {
16          components.add( e:p);
17      }
18
        public double getCost() {
20          double cost = 0.0;
21          for (Component c : components) {
22              cost = cost + c.getCost();
23          }
24          return cost;
25      }
26
        public int size() {
28          int sz = 0;
29          for (Component c : components) {
30              sz = sz + c.size();
31          }
32          return sz;
33      }
34  }
```

# Using Part and Assembly subclasses

```
11  public class Inventory {
12
13      public static void main(String[] args) {
14          // create objects using 'new' operator and appropriate constructor
15          CatalogueEntry screws = new CatalogueEntry( nm:"screw", num:1234, c:0.02);
16          CatalogueEntry nails = new CatalogueEntry( nm:"nail", num:1235, c:0.015);
17
18          Component screw1 = new Part( c:screws);
19          Component screw2 = new Part( c:screws);
20          Component nail1 = new Part( c:nails);
21
22          Assembly a1 = new Assembly();
23          Assembly a2 = new Assembly();
24          a1.add( p:screw1);
25          a2.add( p:screw2);
26          a2.add( p:nail1);
27          a1.add( p:a2);
28
29          System.out.println("Assembly a1 has " + a1.size() + " parts");
30
31
32      }
33  }
```

# Interchangeable objects

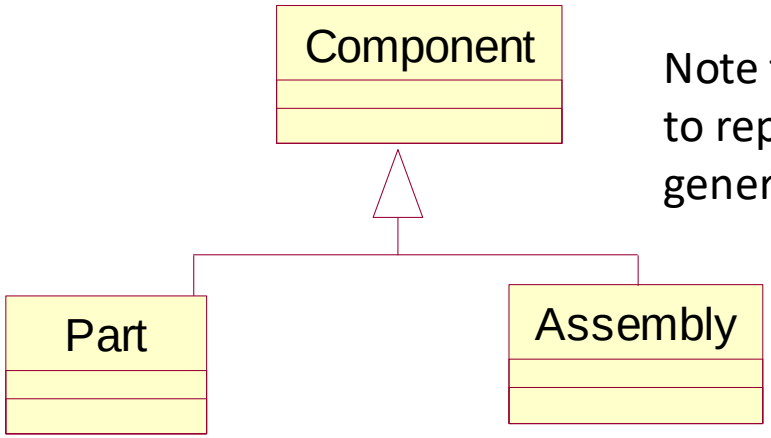
- Since Parts are specialized forms of Components, so are Assemblies
- The variable component has a compile-time type of Component. But at run-time, the type of the objects could be any of the subclasses of Component. The compile-time type can be different, therefore, to the run-time type, giving rise to polymorphism

```
9   public class Assembly extends Component {
        private List<Component> components;
11
12      public Assembly() {
            components = new ArrayList<Component>();
14      }
15      public void add(Component p) {
16          components.add( e:p);
17      }
18
```

Any value that is of Component type can be passed as an argument to add()

# Generalization

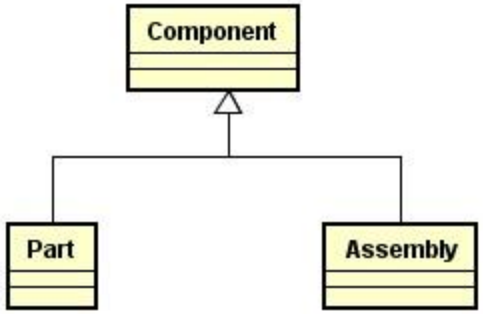
- In UML, the generalization relationship corresponds to public inheritance between classes in C++
- Generalization between classes is shown on class diagrams, since it is a static relationship between classes



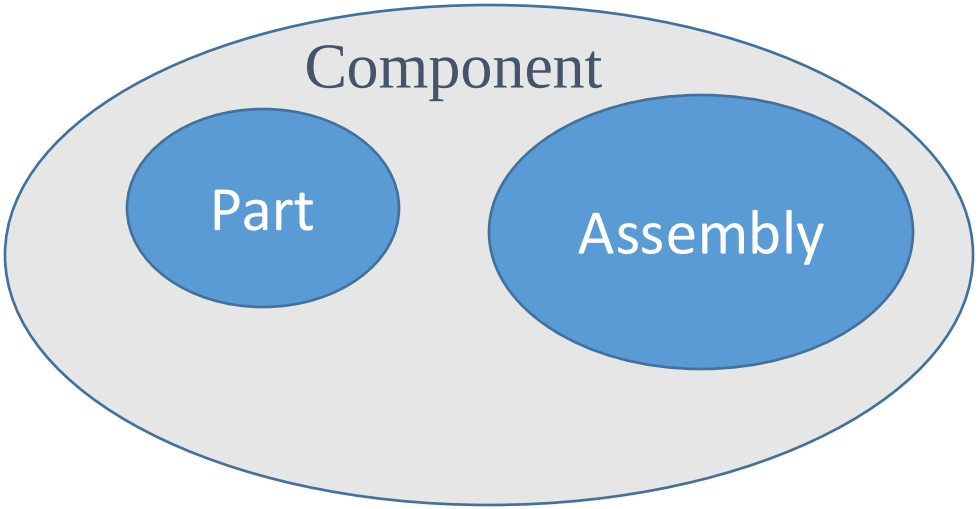

Note the type of arrow used to represent the generalization relationship

- Generalisation is simply a statement that something (e.g., Component) is a more general form of something else (e.g, Part) – the inverse statement is just that Part is a specialised kind of Component

# Another way to look at generalization

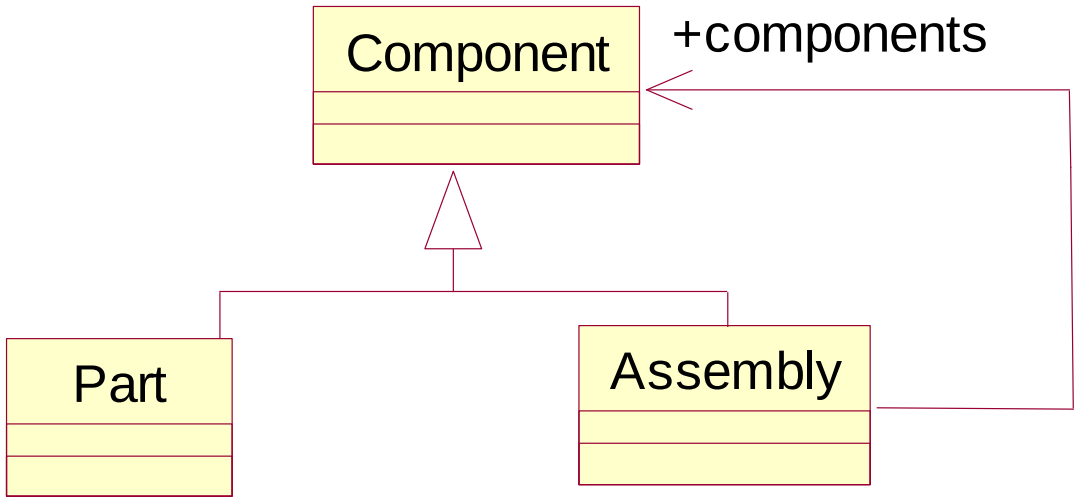



- All Parts are Components
- All Assemblies are Components
- I.e.,
  - $Part \subseteq Component$
  - $Assembly \subseteq Component$



# Polymorphism in the Inventory example

- Polymorphism is shown implicitly in UML diagrams involving objects – for example, by the fact that Assembly objects can be linked to Part objects and other Assembly objects at the same time
- On class diagrams, we can document this polymorphic behaviour by drawing an association from Assembly to the Component class, i.e., to the superclass of Assembly and Part



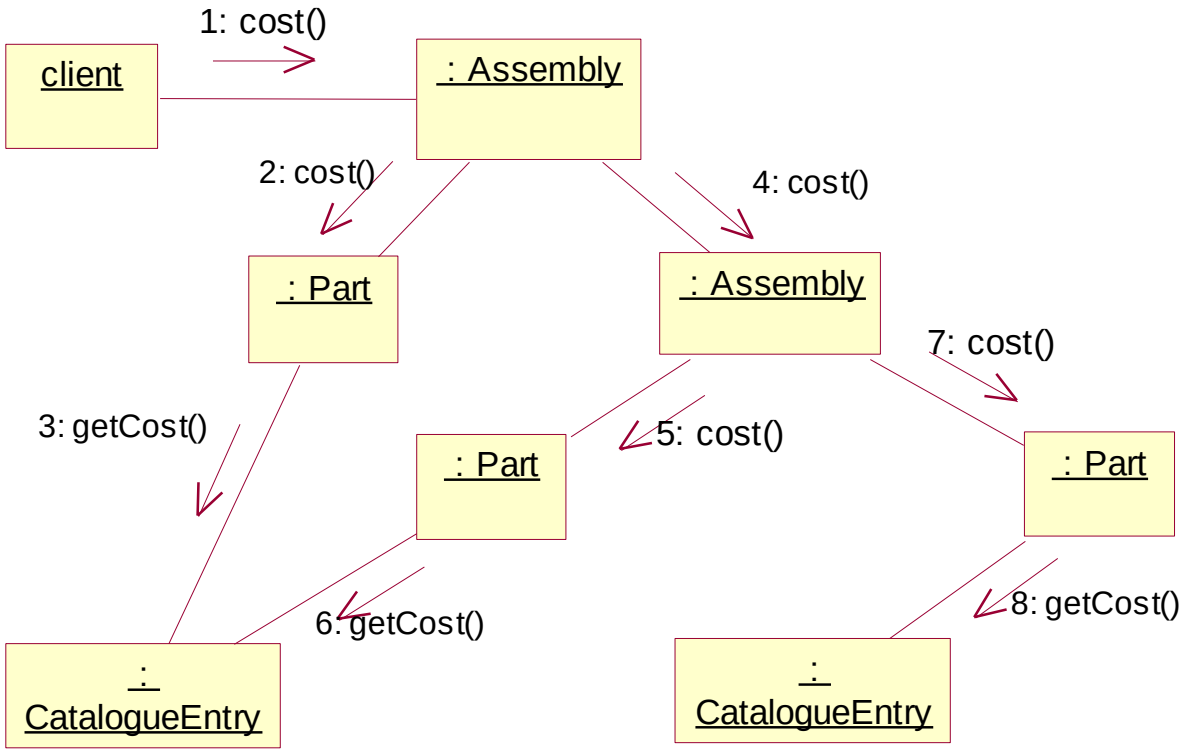
# Specialising behaviour

- What should happen when we ask a Component for its cost?

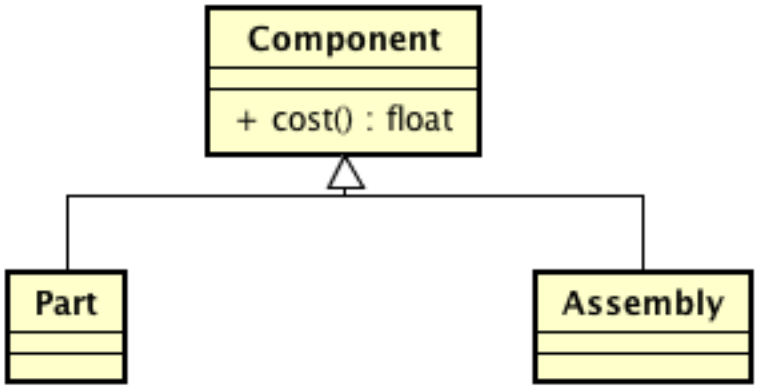
```
11  public class Inventory {
12
13      public static void main(String[] args) {
14          // create objects using 'new' operator and appropriate constructor
15          CatalogueEntry screws = new CatalogueEntry( nm:"screw", num:1234, c:0.02);
16          CatalogueEntry nails = new CatalogueEntry( nm:"nail", num:1235, c:0.015);
17
18          Component screw1 = new Part( c:screws);
19          Component screw2 = new Part( c:screws);
20          Component nail1 = new Part( c:nails);
21
22          Assembly a1 = new Assembly();
23          Assembly a2 = new Assembly();
24          a1.add( p:screw1);
25          a2.add( p:screw2);
26          a2.add( p:nail1);
27          a1.add( p:a2);
28
29          printCost( c:screw1);
30          printCost( c:a1);
31      }
32
33      static void printCost(Component c) {
34          System.out.println("Component cost is " + c.getCost());
35      }
36  }
```

# Polymorphism and Messages

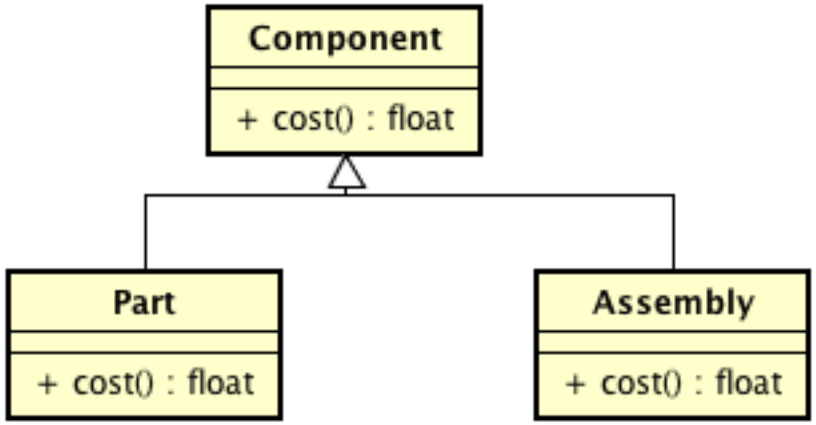
- In a polymorphic data structure, such as an assembly, sending the same message to different objects can lead to different behaviour. For example, if an Assembly wants to find out its cost, it needs to send the cost message to all of its components
  - What each component does when it receives the cost message depends on whether it is a Part or an Assembly object



# Overriding in class hierarchies




No overriding – Part and Assembly use version of cost() inherited from Component




Overriding– Part and Assembly use their own versions of cost()

# Late Binding

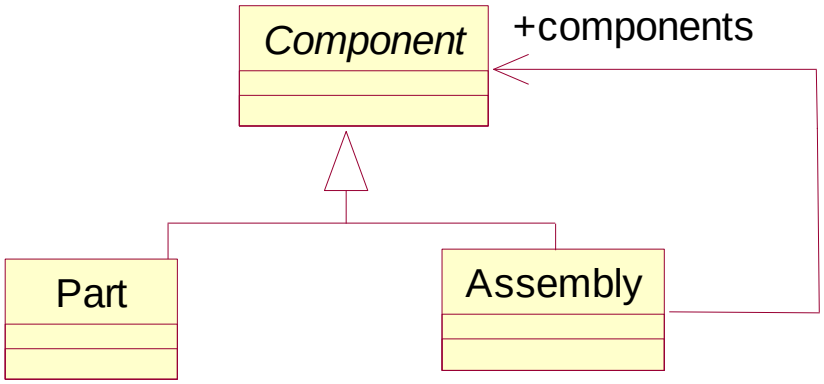
- When the type of the object accessed through a variable can change at run-time, we need different functions to be called at different times
  - The actual function to be called will depend on the type of the accessed object at the point when the function is called.
  - Making the decision about which actual function to call at run-time is called Late (or Dynamic) Binding
- E.g.
  - PrintCost function
  - Compile-time type of parameter is Component
  - But at run-time we may pass an Assembly object
  - Defer decision of which class's version of Cost to call until run-time, when we know the actual type of the object => LATE BINDING of function name to actual code for function

# Static type = contract

- The static type of a value determines what you can do with it
- We can only expect those behaviours that are part of the contract represented by the static type
- The dynamic type (actual type of an object) doesn't matter – as far as what behaviours can be expected
  - But does matter for their implementation

# Abstract Classes

- In our example, we only ever create Part and Assembly objects. We don't create Component objects. The component class is there only to describe the commonality between the Part and Assembly classes, for example
  - That they can both be stored in assemblies
  - That they can both be asked for their cost
- The Component class is an example of an Abstract class
  - In C++, an abstract class is one with at least one pure virtual function
  - In UML, we show that a class is abstract by writing the class name in *italic* font



# Generalization and Inheritance

- The object model distinguishes between generalization and inheritance
- Generalization is about having things in common
- Inheritance is a mechanism for making sure they have things in common
  - Because structure and behaviour of superclass are inherited by subclass