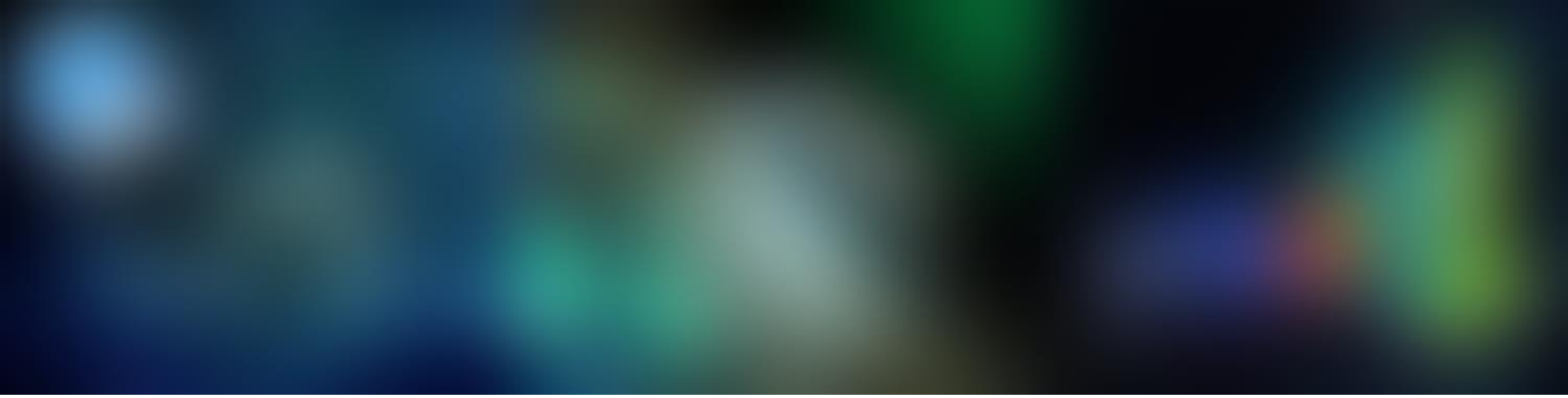
Object-Oriented Programming In Mechatronic Systems

Summer School

Module 5
Aachen, Germany, August 9th, 2018

Cybernetics Lab IMA & IfU
Faculty of Mechanical Engineering
RWTH Aachen University



Closing Java

A few more interesting Java concepts

Closing Java: Static

The keyword `static`

- `static` lets you write or read fields without creating an object first
- Hence, it denotes that a field or method can be accessed without instantiation
- You can call a method by just knowing the class - it means "*behavior not depended on an instance variable*"

Please note:

- A static method can not refer to any instance variable of the class – it is not known which instance variable to use (because it is not specified)
- `static` methods can not use non-static methods, either – because no instance is given
- You can call static methods of an instance (but the method still belongs to the class)

The main method is by convention always static!

Instance Counter

static field (each instance share the same class-variable)

```
public class InstanceCounter
{
   private static int count = 0;

   public InstanceCounter()
   {
      count++;
   }

   public String toString()
   {
      return count + " instances have been created";
   }
}
```

Instance Counter

```
public class Application
{

   public static void main(String[] args){
   {
      System.out.println(new InstanceCounter());
      System.out.println(new InstanceCounter());
   }

}
```

Output:

```
1 instances have been created
2 instances have been created
```

Closing Java: Static

The keyword `static` in combination with final

Have a look at the Java `Math` API!

```
public static final double E = 2.718281828459045235 4;
public static final double PI = 3.14159265358979323846;
```

The Math API is a good example for static methods as well!

`static double`	**`floor`**`(double a)` Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.
`static int`	**`getExponent`**`(double d)` Returns the unbiased exponent used in the representation of a `double`.
`static int`	**`getExponent`**`(float f)` Returns the unbiased exponent used in the representation of a `float`.
`static double`	**`hypot`**`(double x, double y)` Returns sqrt($x^2 + y^2$) without intermediate overflow or underflow.
`static double`	**`IEEEremainder`**`(double f1, double f2)` Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
`static double`	**`log`**`(double a)` Returns the natural logarithm (base *e*) of a `double` value.
`static double`	**`log10`**`(double a)` Returns the base 10 logarithm of a `double` value.
`static double`	**`log1p`**`(double x)` Returns the natural logarithm of the sum of the argument and 1.
`static double`	**`max`**`(double a, double b)` Returns the greater of two `double` values.
`static float`	**`max`**`(float a, float b)` Returns the greater of two `float` values.

Math-Class: Using static fields and methods (an example)

```
public class Circle {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calcArea(){
        return Math.PI * Math.pow(r, 2.0);
    }

    public double calcCircumference(){
        return 2.0 * Math.PI * radius;
    }

}
```

static method `pow`

static constant field `PI`

Implementation of an Exemplary Application

Enterprise Resource Planning (ERP)

Enterprise Resource Planning (ERP) refers to the entrepreneurial task of **planning** and **controlling resources** such as **capital, personnel, resources, materials, information** and **communication technology** and **IT systems** in a timely and demand-oriented manner for the company's purpose. The aim is to ensure an efficient operational value-added process and constantly optimized control of corporate and operational processes.

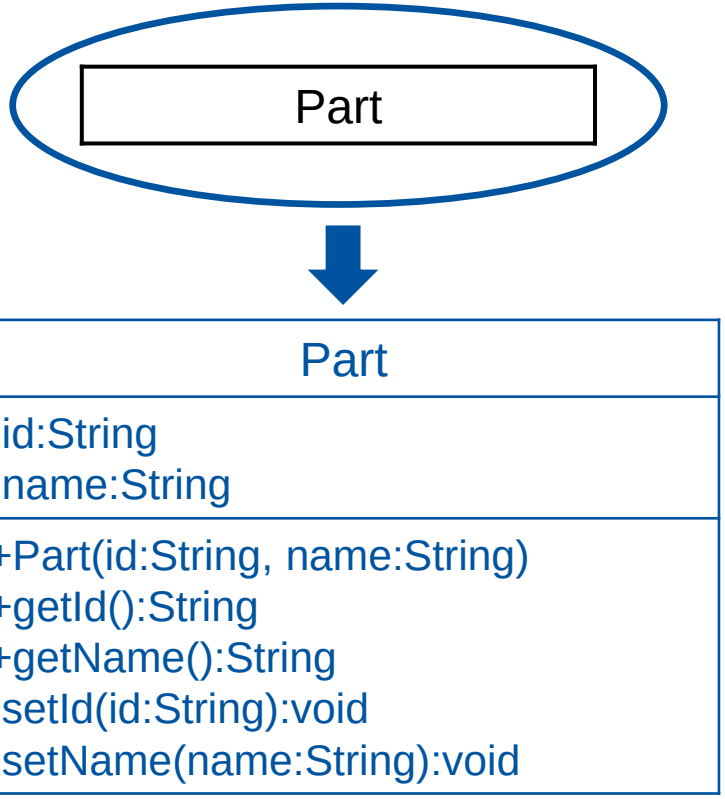
One of the core functions of ERP in **manufacturing companies** is **material requirements planning**, which must ensure that all materials required for the manufacture of **products** and **parts** are available in the **right place**, at the **right time** and in the **right quantity**.

We focus on this core function and start by implementing some core functions of such an system!

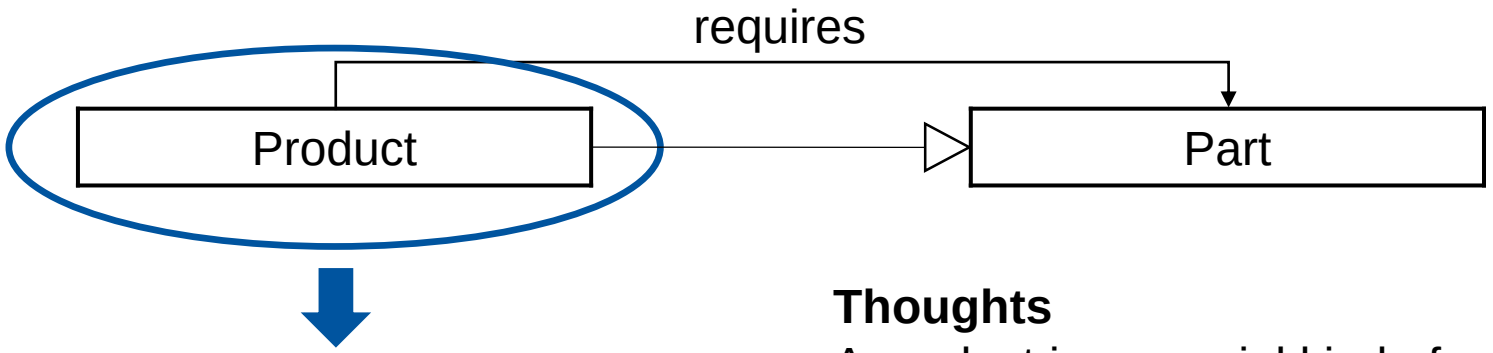
Parts and Products

Thoughts
Each part in our system should be identifiable by a **unique id**. Further, it should be possible to describe each part by a meaningful **name**.

We can implement this very easily!



Parts and Products



Product
-requirements:**PartRequirementList**
+Product(id:String, name:String) …

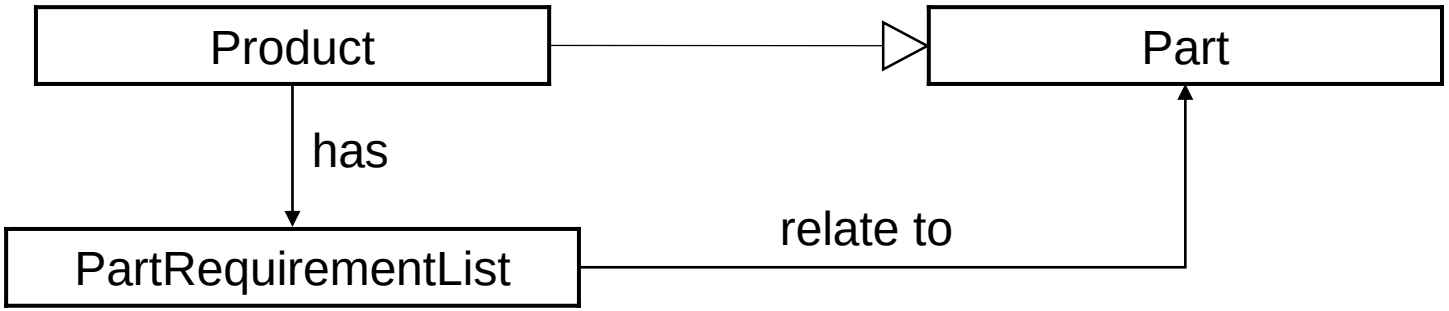
Thoughts

A product is a special kind of a part. We require a set of parts to manufacture the product. But it could be also a part. Hence, we need to define the required parts to manufacture our product.

Some more work to do before implementation!

Part Requirements

Thoughts
A product requirement consists of two parts: the required part (type) and the number of parts (quantity).
Hence: Product requirements are a list of parts and their corresponding quantity!

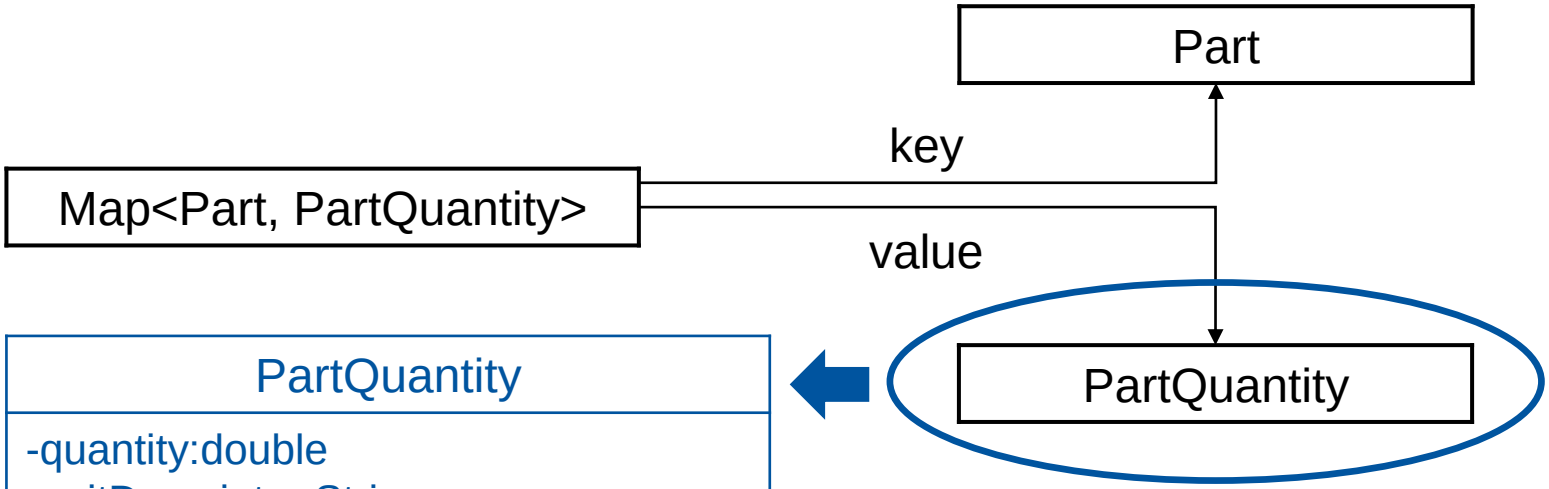


How do we define these requirements?

Part Quantities

Thoughts

We need a data structure to store the part and the corresponding part quantity. The part is the marking element (key) and the quantity is the related information (value)!



… and toString()

We can implement PartQuantity very easily!

Excurse: `toString()`-Method

What is the output if you implement something like that:

```
System.out.println(new PartQuantity("g", 100));
```

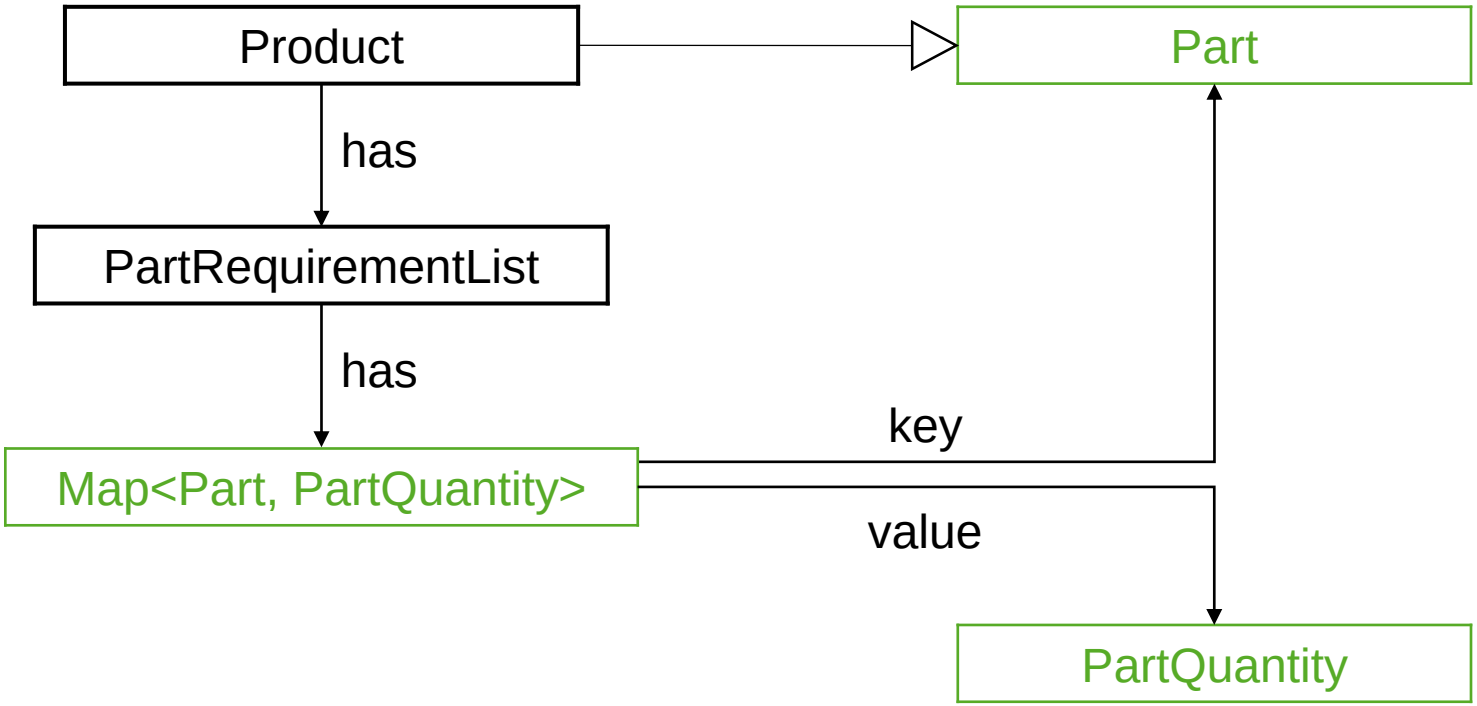
➡ `de.ima.opms.erp.example.model.PartQuantity@15db9742`

Overriding the toString():String method allows you to define the translation of an object into a String:

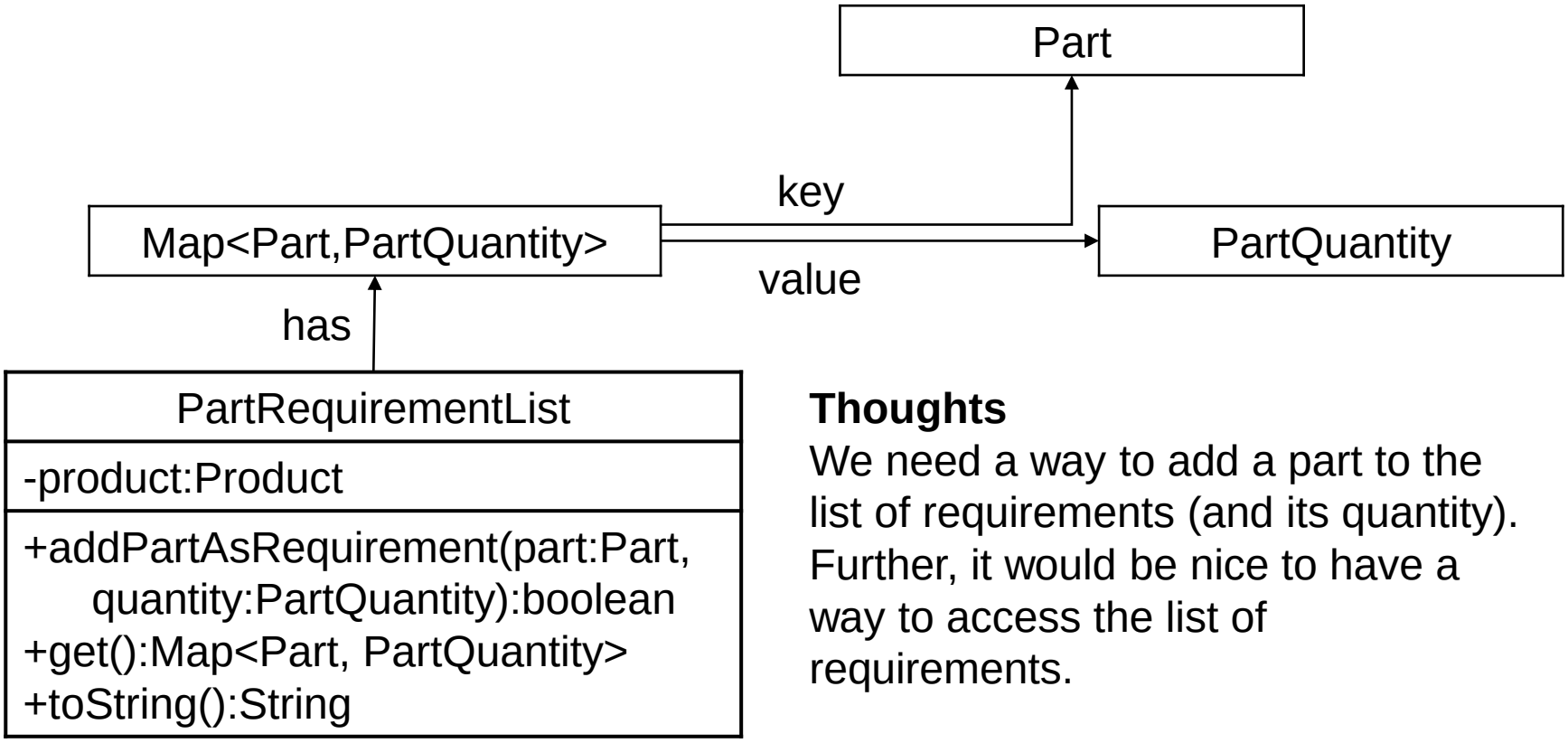
```
public String toString() {
   return getQuantity() + (getUnitDescriptor() != null ?
      getUnitDescriptor() : "");
}
```

➡ `100.0g`

Interim Conclusion of our class modelling



Part Requirements in Detail



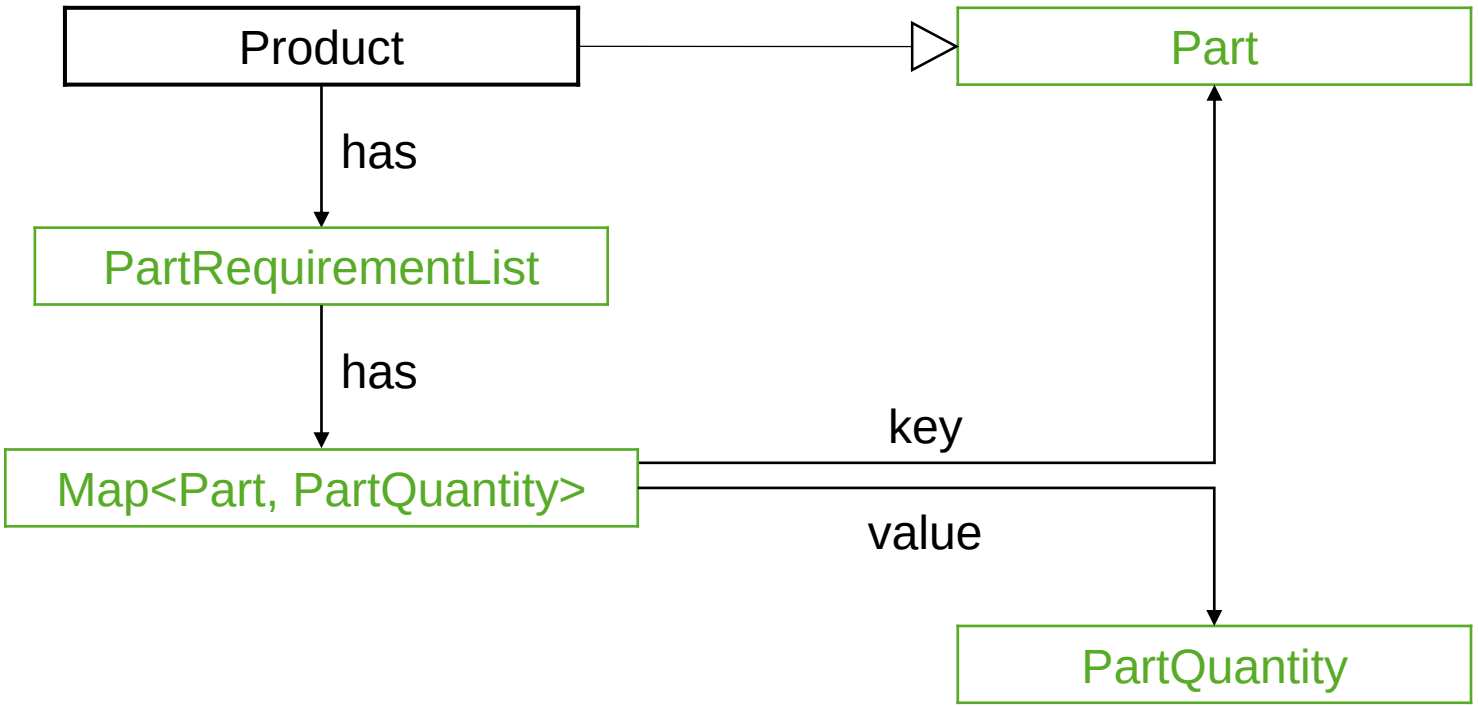
Thoughts

We need a way to add a part to the list of requirements (and its quantity). Further, it would be nice to have a way to access the list of requirements.

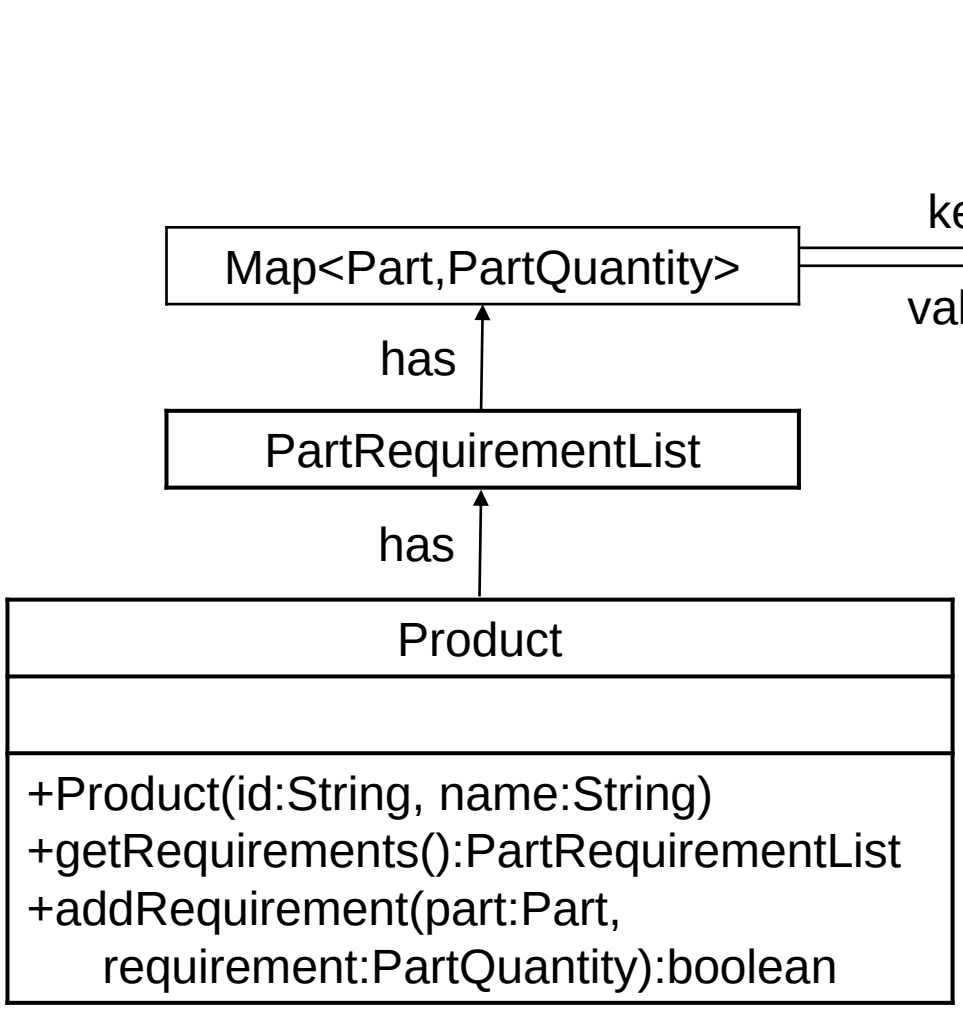
So lets do the first part of the implementation!

Implementing Product

Finally, we can implement a first version of our product!



Products in Detail



Thoughts

A product is a more specialized part and has some requirements.

So lets do the next part of the implementation!

Warehousing

Besides the products and parts, we need something to store everything

➡ Warehouse

Thoughts

Our warehouse needs to have a stock of parts (and products). For each part and product we need to manage the stored quantity. Further, we should have some methods to check if a part is available as well as to stock in and out.

Warehouse
-stock:Map<Part,PartQuantity>
+Warehouse() +isAvailable(part:Part, quantity:PartQuantity):boolean +isAvailable(partList:Map<Part, PartQuantity>):boolean +stockIn(part:Part, quantity:PartQuantity):void +stockOut(partList:Map<Part, PartQuantity>):void +stockOut(part:Part, quantity:PartQuantity):void

Warehousing

Warehouse
-stock:Map<Part,PartQuantity>
+Warehouse() **+isAvailable(part:Part, quantity:PartQuantity):boolean** +isAvailable(partList:Map<Part, PartQuantity>):boolean +stockIn(part:Part, quantity:PartQuantity):void +stockOut(partList:Map<Part, PartQuantity>):void +stockOut(part:Part, quantity:PartQuantity):void

We need to check if the available stock satisfies the needed quantity!

1. **Get** the **available quantity**!
2. **Check** if a **quantity** is set for the part, if not return false. Otherwise (else), check if the **unit descriptors are compatible** (easy mode).
3. If the descriptors are **not** compatible, **throw an unsupported operation exception**. **Otherwise**, check if the **available quantity is larger or equal to the request one**.

Warehousing

```
public boolean isAvailable(Part part, PartQuantity quantity)
   throws UnsupportedOperationException {

   // Get the available quantity!
   PartQuantity availableQuantity = stock.get(part);

   /* Check if a quantity is set for the part, if not return false. Otherwise,
      check if the unit descriptors are compatible */
   if (availableQuantity == null) {
      return false;
   }
   checkCompatibleUnitDescriptors(availableQuantity,
      quantity, true);

   // Otherwise, check if the available quantity is larger or equal to the
      request one.
   return availableQuantity.getQuantity() >=
      quantity.getQuantity();
}
```

Warehousing

```
private boolean checkCompatibleUnitDescriptors(PartQuantity
   quantity1, PartQuantity quantity2, boolean throwException)
   throws UnsupportedOperationException {

   boolean result = (quantity1.getUnitDescriptor() == null &&
      quantity2.getUnitDescriptor() == null) ||
      quantity1.getUnitDescriptor().
         equals(quantity2.getUnitDescriptor());

   if (!result && throwException) {
      throw new UnsupportedOperationException(
         "Quantity descriptors are unequal [" +
         quantity1.getUnitDescriptor() + "|" +
         quantity2.getUnitDescriptor());
   }

   return result;
}
```

Warehousing

Warehouse
-stock:Map<Part,PartQuantity>
+Warehouse() +isAvailable(part:Part, quantity:PartQuantity):boolean +isAvailable(partList:Map<Part, PartQuantity>):boolean +stockIn(part:Part, quantity:PartQuantity):void +stockOut(partList:Map<Part, PartQuantity>):void +stockOut(part:Part, quantity:PartQuantity):void

Let us implement our warehouse – step by step!

Interim Conclusion II

We have everything ready to test our implementation in a first scenario!

Product: Apple Pie

Part list:

4 egg (s)
250 g sugar
125 g butter
100 ml milk
300 g flour
3 tsp. baking powder
5 m. -size apples

Thank you very much!

IfU IMA RWTH AACHEN UNIVERSITY