



Object-Oriented Programming In Mechatronic Systems

Summer School 2018

Module 5 – Implementing Real-World Applications with Java
Aachen, Germany

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



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.

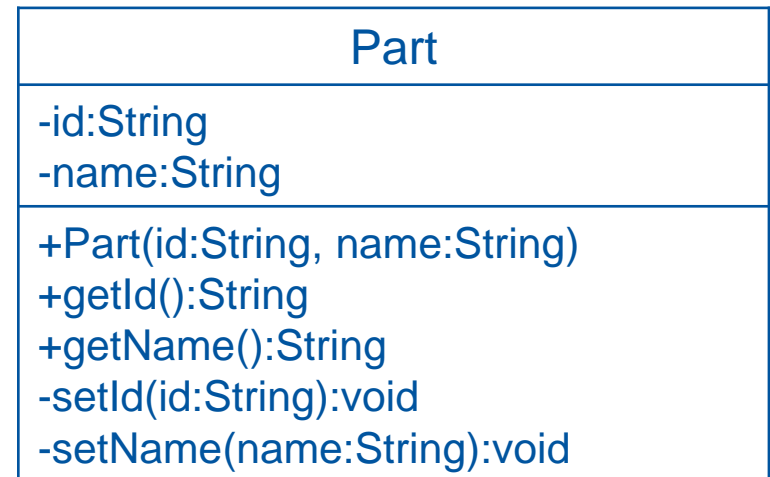
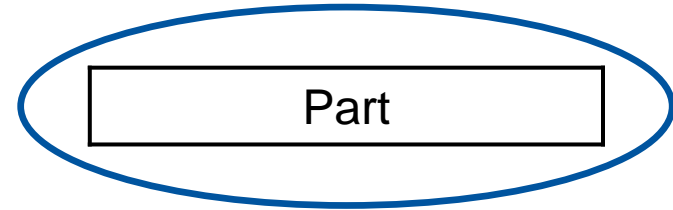
One of the core functions of ERP in **manufacturing companies** is **material requirements planning**, which must ensure that all materials required for the manufacture of **products** and **parts** are available in the **right place**, at the **right time** and in the **right quantity**.

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

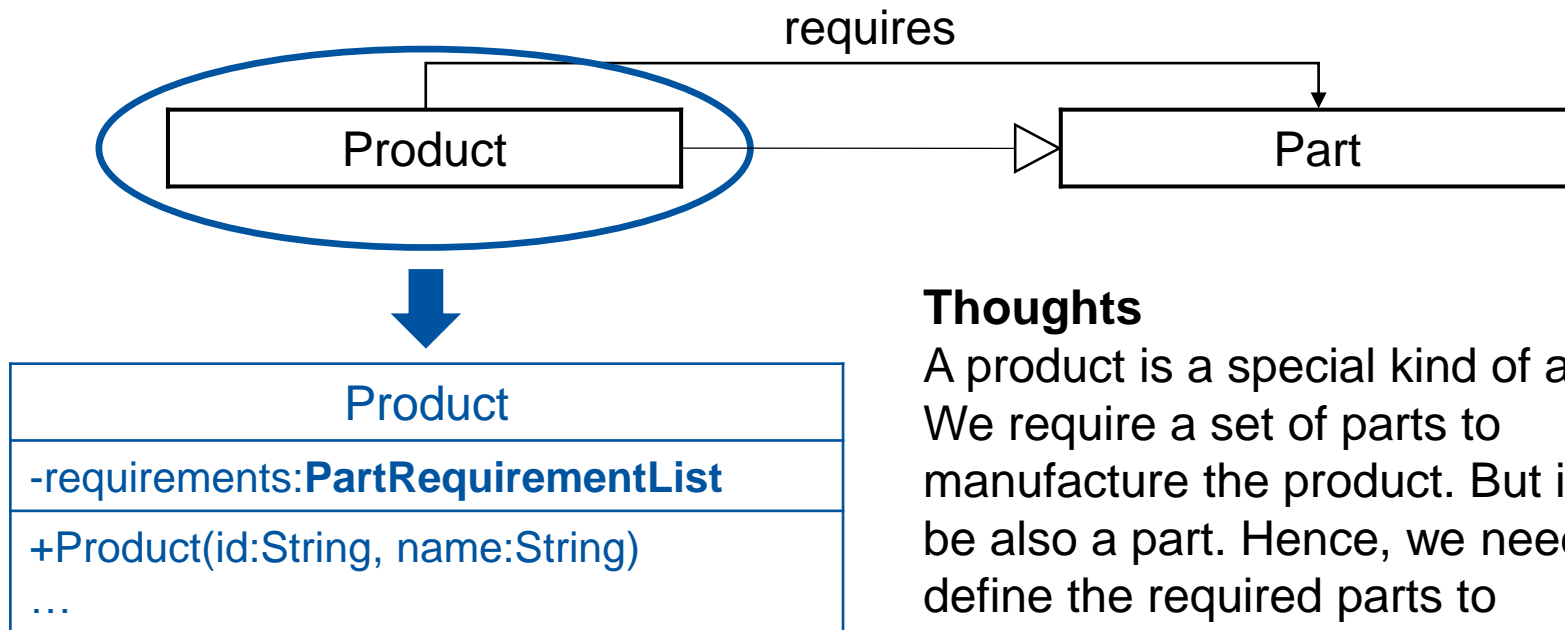
Thoughts

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



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.

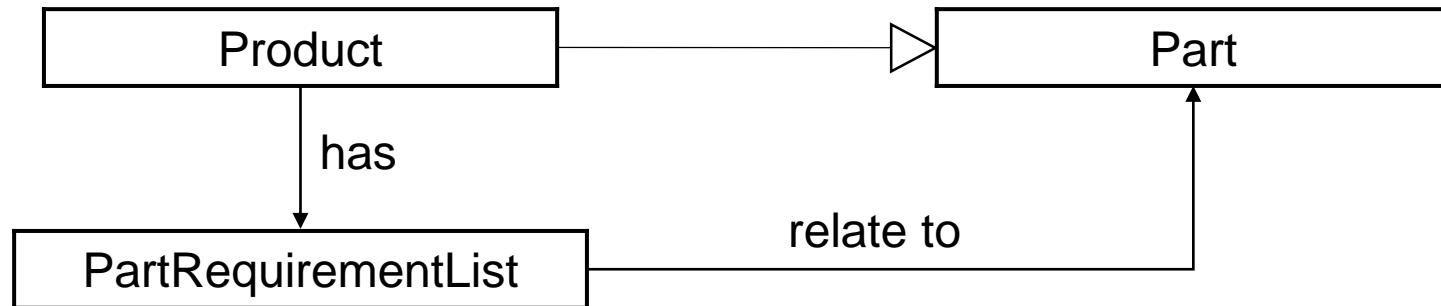
Some more work to do before implementation!

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!

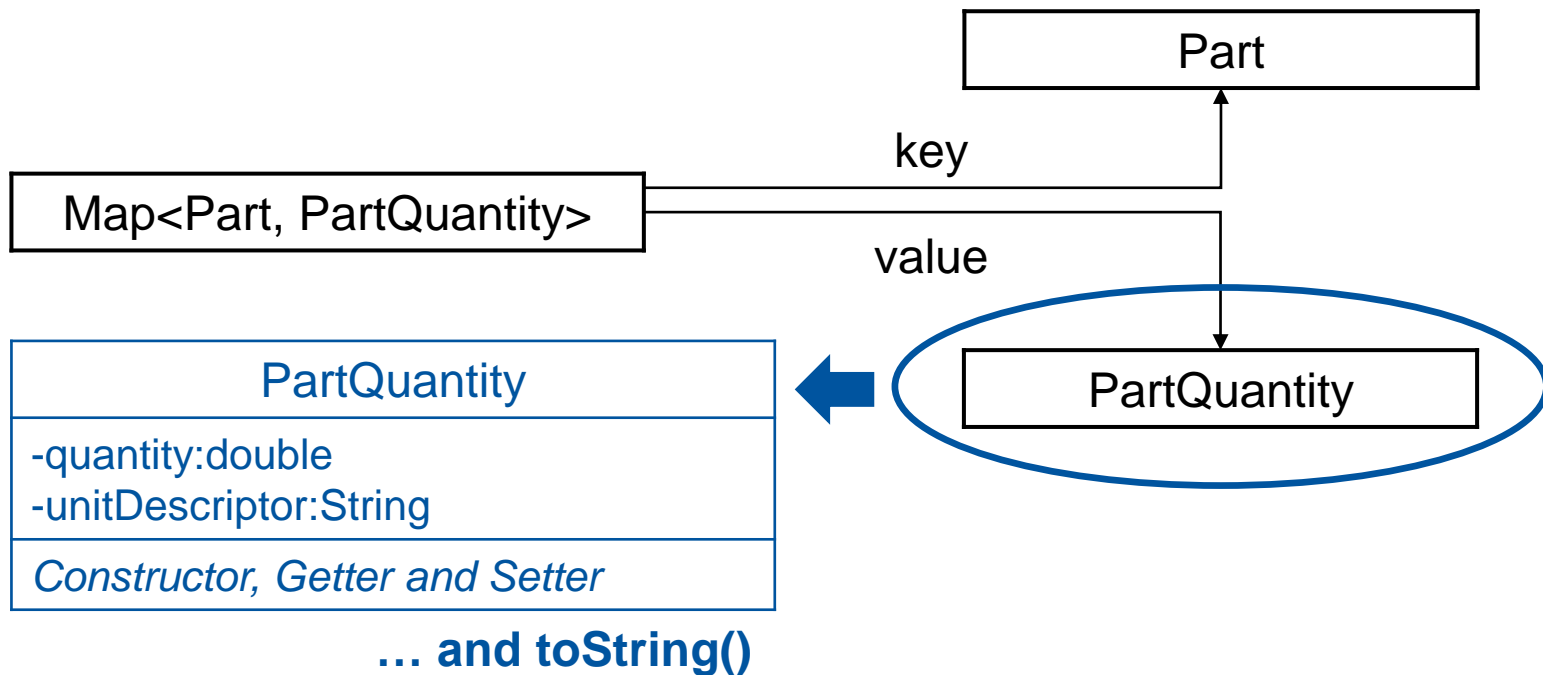


How do we define these requirements?

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)!



We can implement PartQuantity very easily!

Excuse: toString()-Method

What is the output if you implement something like that:

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

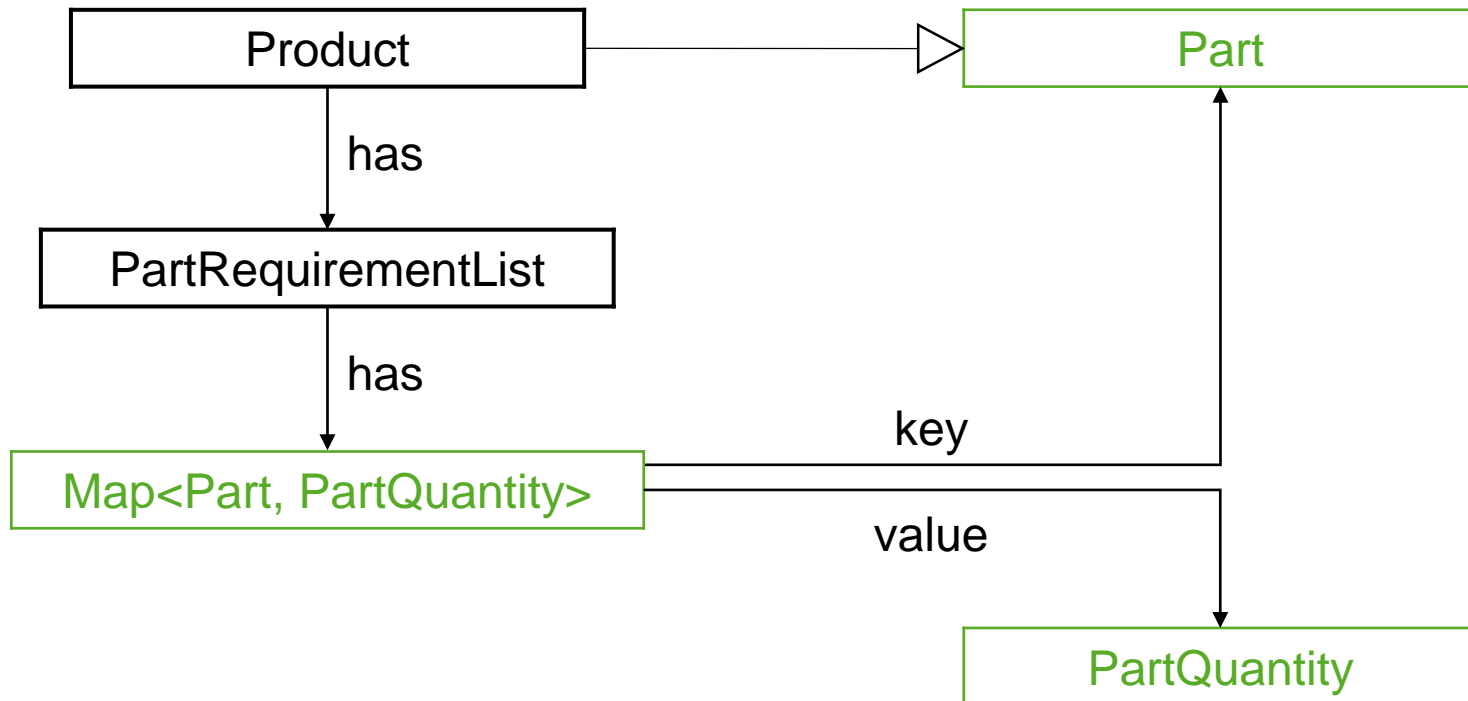
➔ `de.ima.opms.erp.example.model.PartQuantity@15db9742`

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

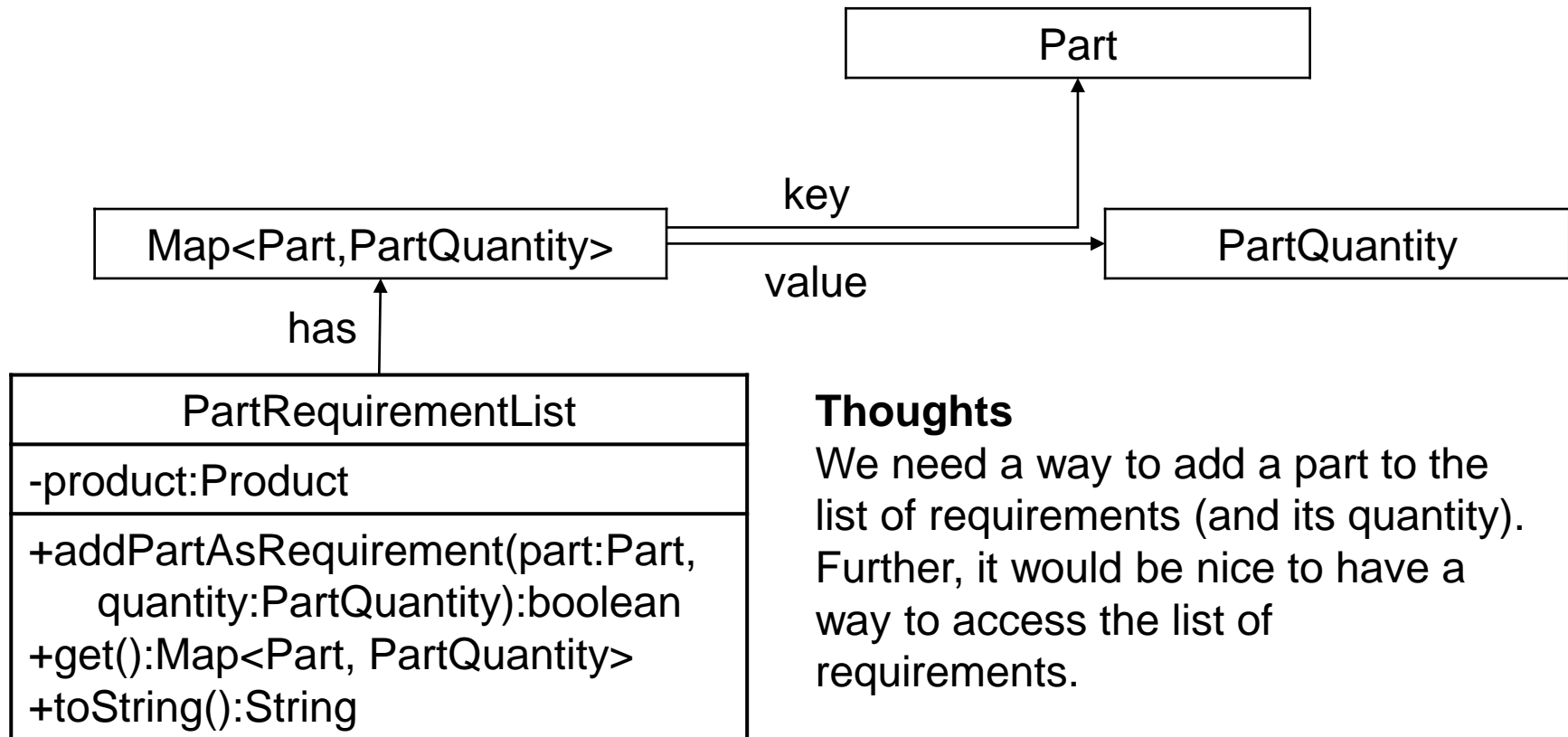
```
public String toString() {  
    return getQuantity() + (getUnitDescriptor() != null ?  
        getUnitDescriptor() : "");  
}
```

➔ `100.0g`

Interim Conclusion of our class modelling



Part Requirements in Detail



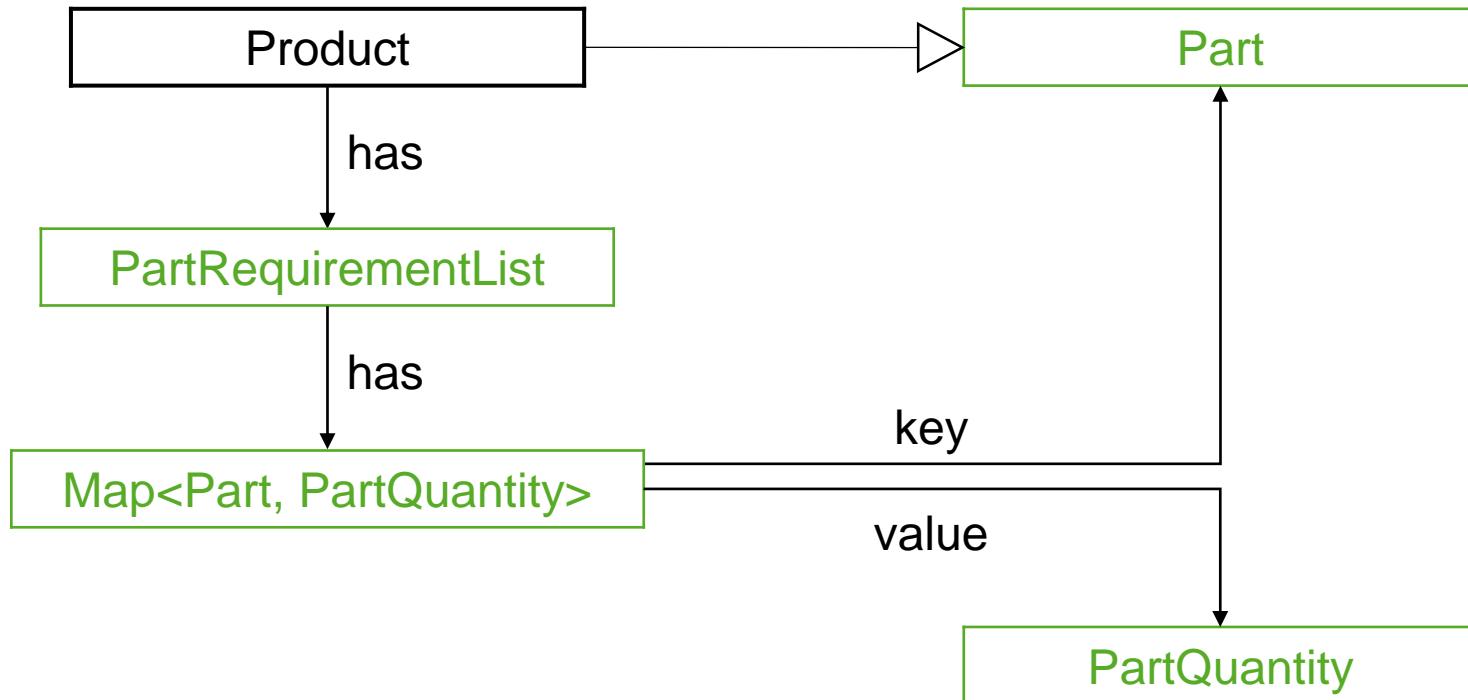
Thoughts

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.

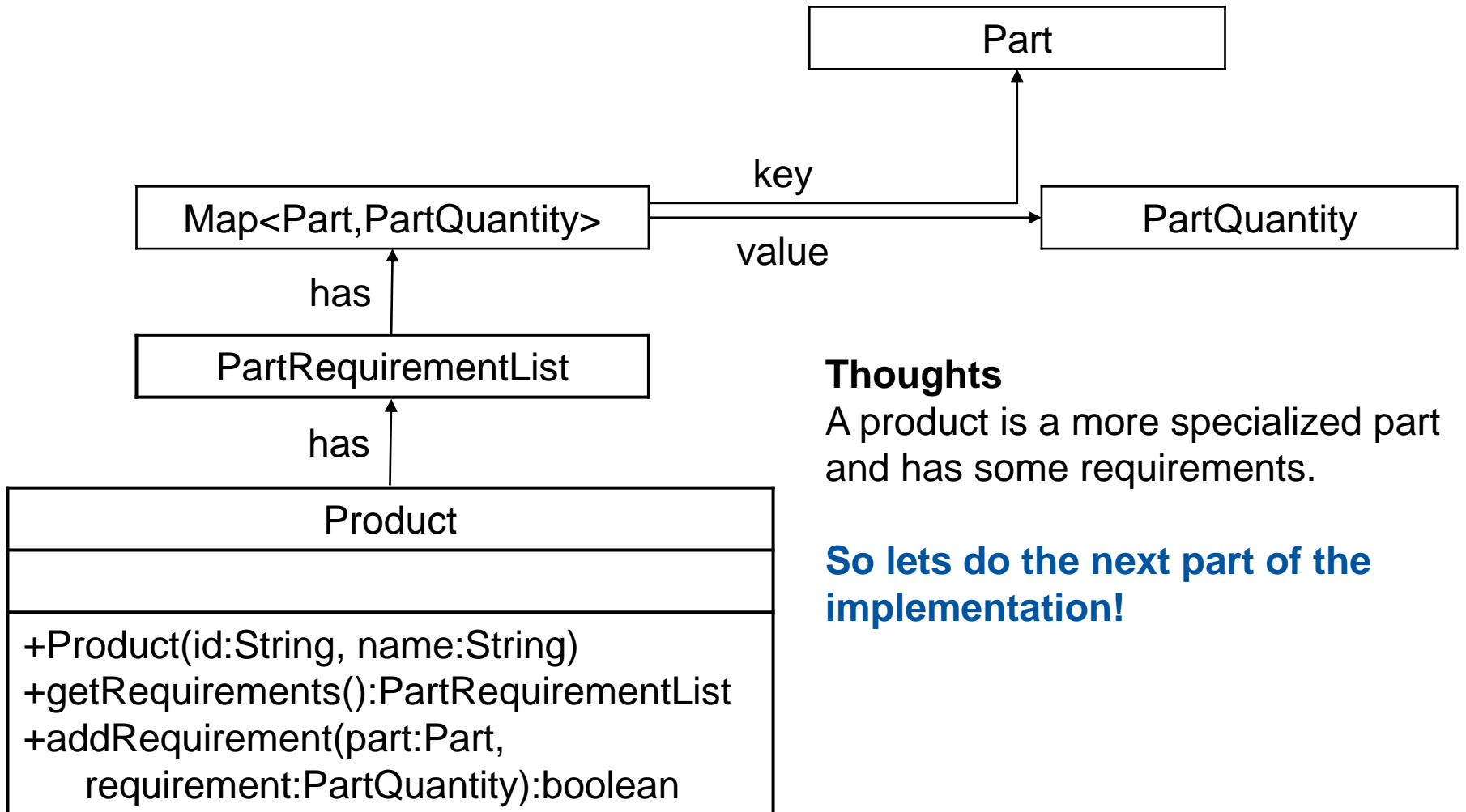
So lets do the first part of the implementation!

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!

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;
}
```


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!

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!