

# Final Project for Computer Science, EOOP 2020

Krzysztof Rudnicki, 307585

June 11, 2020

# Chapter 1

## Problem and general design

### 1.1 Problem

My goal was to design and implement *UI*, *tests* and *classes* with methods for a project using keyword *Warehouse* in C++. Most challenging in classes design was coming up with methods that were both useful and relatively easy to implement. Tests were pretty natural to create after coming up with ideas of classes. However I couldn't come up with any UI, probably because of mindset focused on implementing methods rather than thinking how those methods should be used by the user.

### 1.2 General design

This section is divided into four classes that I used. I wanted to have minimal number of classes and focus on their quality rather than quantity.

#### 1.2.1 Good

##### Attributes

1. Size - int vector, assuming that all the goods are cuboid it describes object *width, length and height*.
2. hazard - bool vector, amount of possible hazards are defined by HAZARD constant (by default 3)
3. stateOfMatter- Type int, number of possible states defined by STATE constant (by default 3 - solid, liquid, gases)
4. weight - Type unsigned int, used to check whether the good is light enough to be transported by some equipment.

## Methods

1. Constructor - Sets all attributes to 0 including vectors, it also sets maximal size of size vector and hazard vector to its corresponding constant.
2. getters and setters - Allow us to get and set every attribute inside the class.
3. volume - calculates volume of the good using Size attribute and formula  $V = W * H * L$  where V - volume, W - width, H - height and L - length

## General comments

Good is by far the easiest class in the project, having just one method outside of getters, setters and constructor. It is also reflected in good.cpp file, having only 85 lines of code, most of it being very short and not using any extra methods.

### 1.2.2 Equipment

#### Attributes

1. canTransportHazard and canTransportState - bool vectors, checking whether the equipment is capable of transporting certain type of hazards and states. Maximal size of those vectors are defined using HAZARDS and STATE constants.
2. sizeLimit and sizeLeft - sizeLimit is the maximal size of the good that can be transported assuming that no good was placed on the equipment before. sizeLeft is sizeLimit minus size of whatever good put on the equipment.
3. trainingRequired - bool vector defining what kind of training does the employee need to use the equipment. Maximal size defined by CERTIFICATES constant (default 10)
4. timesUsed and timesUsedLimit - indicates how much was the equipment used and how much the equipment can be used before it needs repair.
5. weightLimit and weightLeft - weightLimit indicates maximal weight of goods on the equipment before it is overloaded, weightLeft is weightLimit minus sum of weights of all the goods on the equipment.

#### Methods

1. Constructor - Sets all attributes to 0 including vectors, it also sets maximal size of vectors using global constants.
2. getters and setters - Allow us to get and set every attribute inside the class.

3. volumeLeft and volumeLimit - Used more for the sake of user who wants to check how much of the volume is left or how much he can occupy rather to check if equipment can move goods.
4. canMoveGood and "sub" methods - canMoveGood checks using smaller functions which check if equipment can move state, hazard and whether it has enough space for the good.
5. occupySpaceGood - Basically setter for size left but allows us to just put good name and simplify the process.
6. operator= - Important when assigning equipment to the employee.

### **General comments**

Equipment class was harder than good class but still relatively easy. Most complicated method is of course canMoveGood which was divided in three smaller ones following philosophy of "divide-and-conquer"

### **1.2.3 Employee**

#### **Attributes**

1. trainingVectors - Three bool vectors depicting employee ability to move goods in certain states and hazards. Additionally equipmentTraining telling us what certificates does the employee have and therefore what equipment he can use.
2. canRepair - Tells us whether the employee can repair equipment or not.
3. assignedEquipment - Equipment type object, initially planned to used in order to simplify process of moving object.

#### **Methods**

1. Constructor - Sets all attributes to 0 including vectors, it also sets maximal size of vectors using global constants. assignedEquipment is set to equipment created using equipment constructor.
2. getters - Allow us to get every attribute from the class, instead of using classic setters we use train methods and assignEquipment method.
3. train - four methods allowing us to train employee for Hazards, States, Equipment and the ability to repair.
4. canMoveGood and sub methods - Checks if employee has proper training connected with state and hazards of the good. It also checks if employee can use the equipment that is assigned to move the good.

5. moveGood and moveGoodOverride - moveGood is a "safe" version which checks if employee can in fact move good, moveGoodOverride is mostly there for debug reasons and is used in Warehouse class method addGood where we already know that the good can be moved.
6. repairEquipment - if employee can repair equipment, the equipment timeUsed is changed to 0.

### General comments

Employee class has the most relations out of all classes and therefore it is most complicated. It consists crucial method moveGood which is the base for warehouse method shipGoods later on.

### 1.2.4 Warehouse

#### Attributes

1. allEmployees, allEquipment and allGoods - vectors consisting of pointers to employees, equipment and goods classes. Using pointers makes it possible to modify for example employee from the warehouse and be sure that those modifications will take effect in allEmployees vector.
2. size - as before, int vector, assumes the warehouse is a cuboid and holds its length, width and height

#### Methods

1. Constructor - the only attribute that is set up "manually" is size vector which size is limited using DIMENSION constant and all of its elements are set to 0.
2. getters and "adders" - Allow us to get every attribute from the class, there are no classic setters, instead we add one employee, good or equipment every time.
3. capacityLeft and capacityTotal - those methods take all the equipment inside the warehouse and sum its sizeLimit and sizeLeft.
4. volume - calculates volume of the warehouse using Size attribute and formula  $V = W * H * L$  where V - volume, W - width, H - height and L - length
5. shipGoods and its sub methods - by far the most complicated method which relies heavily on working methods canMoveGood for equipment and employee. Flowchart below demonstrates how this method works:

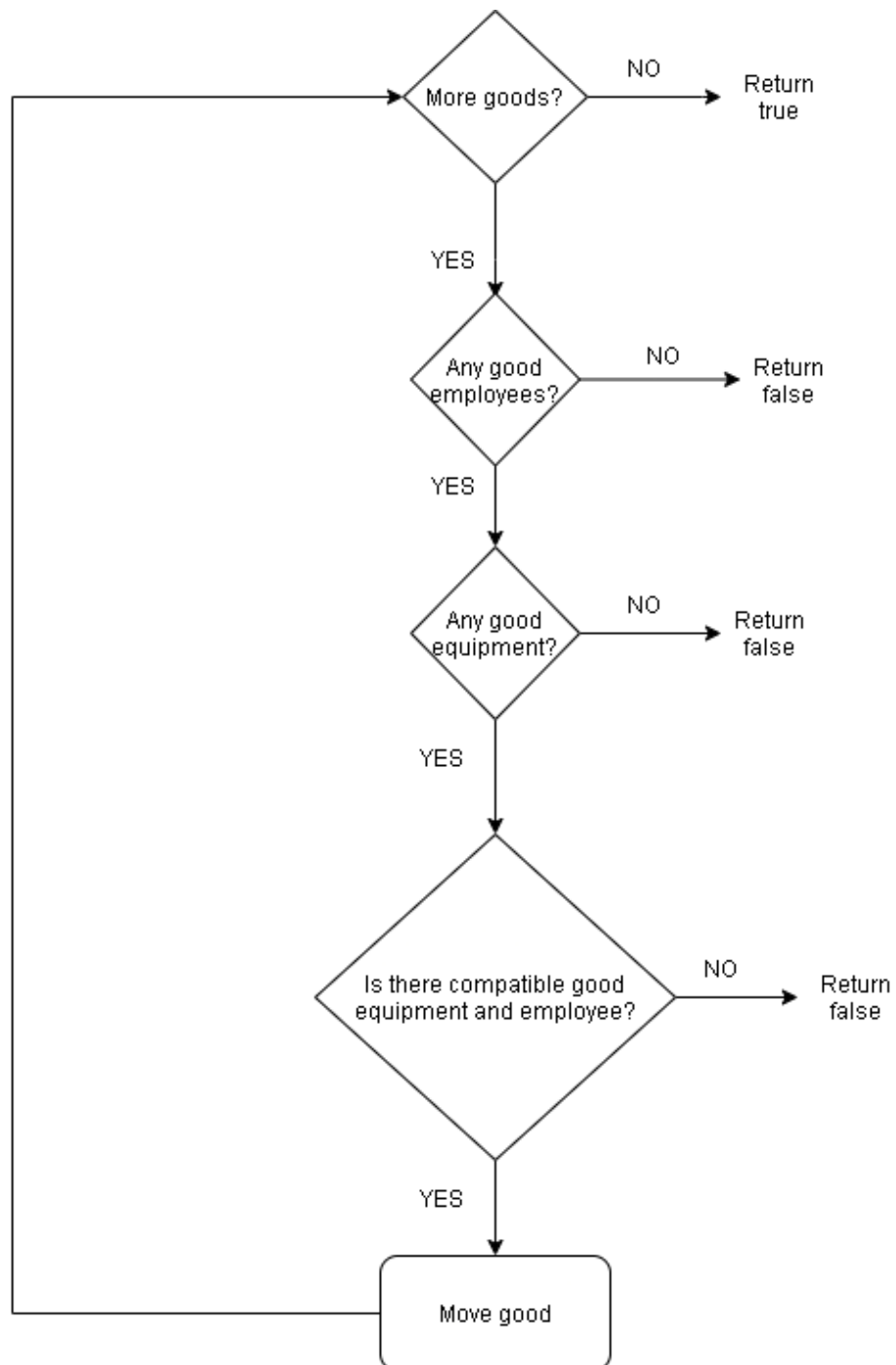


Figure 1.1: shipGoods method flowchart

### **General comments**

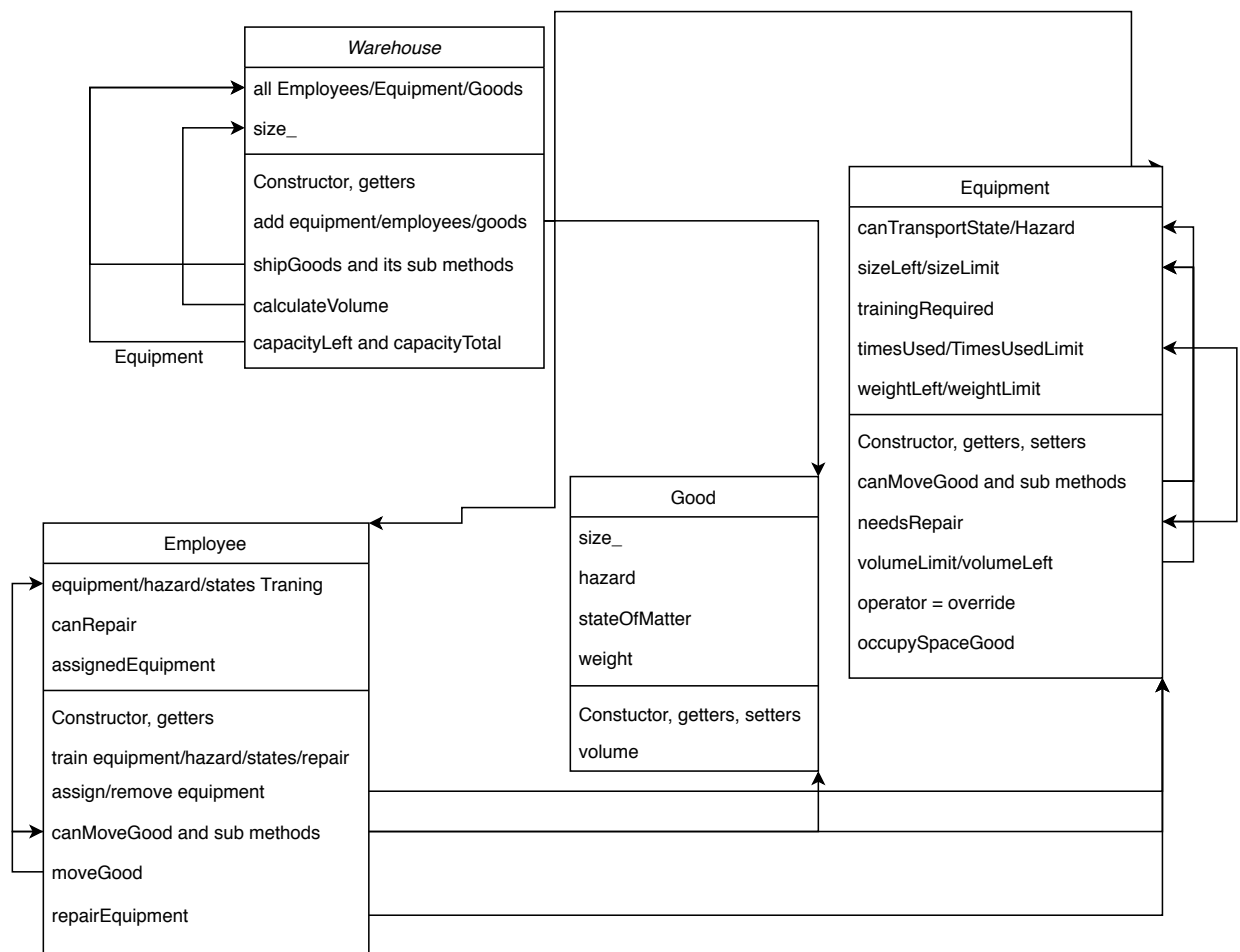
Warehouse was easier to made than one initially can thought because it was reusing most of its code for shipGoods from the methods created in employee and equipment class. One of the challenges was the fact that vectors which hold information about goods, equipment and employees were containing pointers to these objects and therefore most of the methods were wrote differently than in previous classes.





## Chapter 2

## Updated schema



## Chapter 3

# Header files

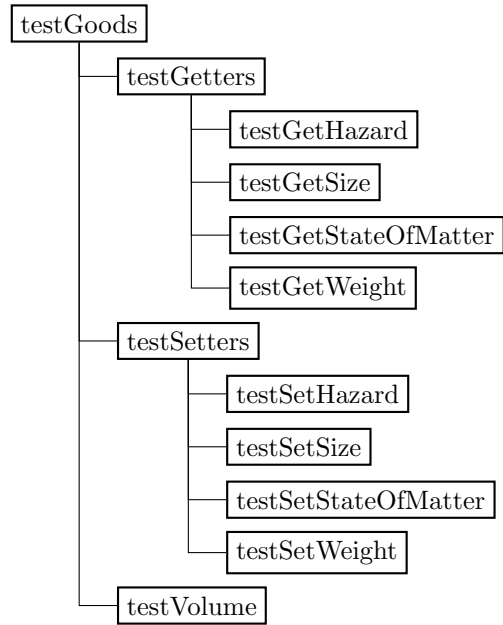
Since all the header files were send alongside the coding part I will exclude them from this report. However differences between the original header files and final ones are available in **Summary** chapter.

## Chapter 4

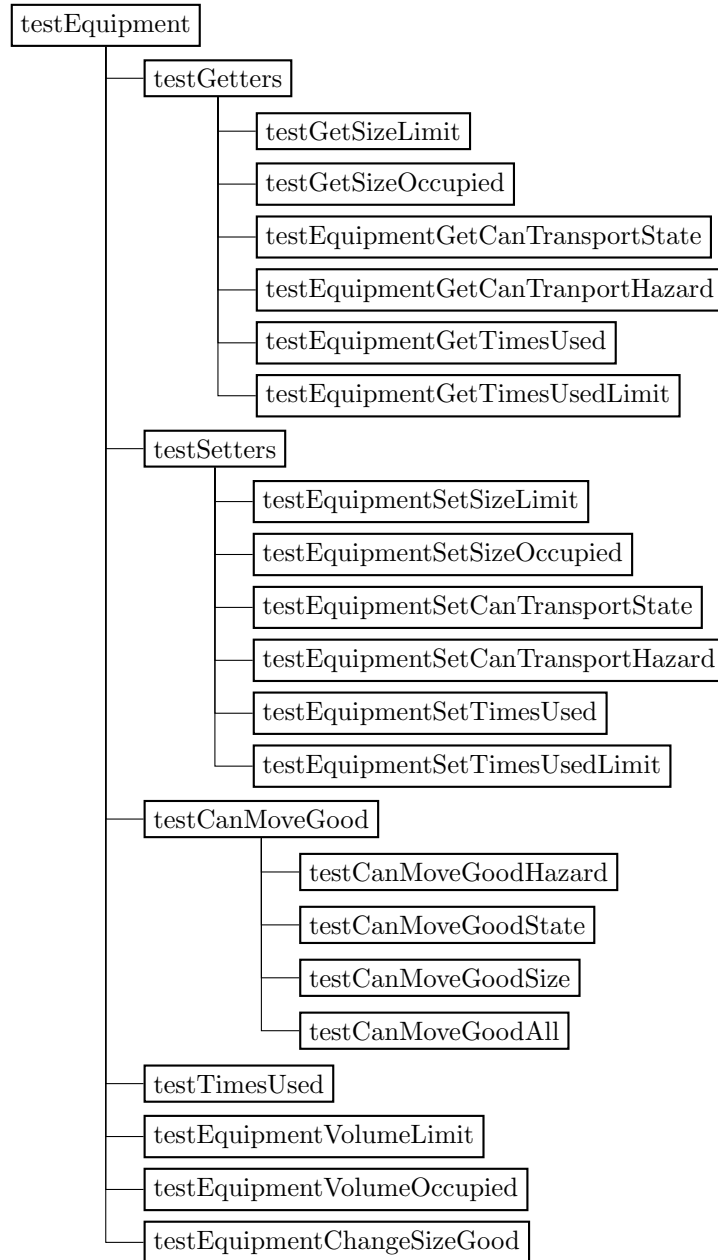
# Summary of tests

There are 57 functions in main.cpp file which start with the word "test", tests are arranged in tree-like structure starting from one named in most generic way for example *testGood* and going deeper and deeper until they check single methods. Below I presented those tree like structures for tests for each class. My goal was to have at least one test for each and every method for every class and I believe that I succeeded in doing so. Tests are the hybrid between `std::cerr` showing only when some test fails and `std::cout` which is used mostly to print the behaviour of the program and manually compare it with expected results. In hindsight I should only use `std::cerr` type of tests as they are much more readable and only show if there was some unexpected behaviour. Using `std::cout` made it harder to spot those errors. Tests are also arranged from the simplest one to the hardest one. No framework was used while making tests.

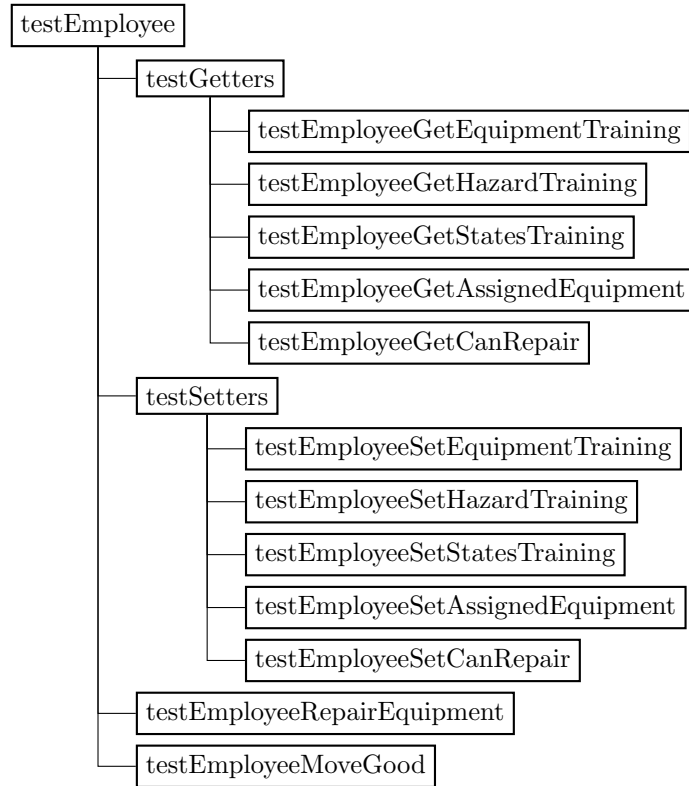
## 4.1 Good Tests



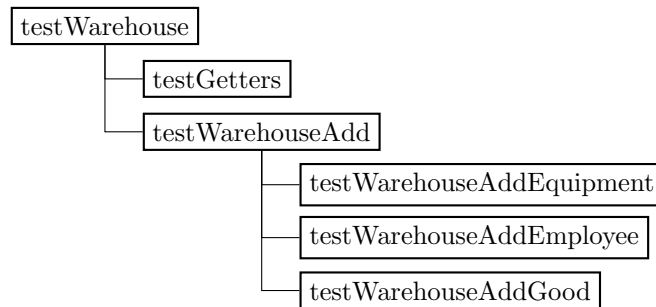
## 4.2 Equipment Tests



### 4.3 Employee Tests



### 4.4 Warehouse Tests



## Chapter 5

# Summary

### 5.1 Changes

### 5.2 Test Changes


Instead of testing four classes in four big functions, those four functions were divided into many many smaller ones which checked single methods. Much more tests were added and most of them were modified after I wrote code for methods that were being tested.

#### 5.2.1 Global changes

##### **Creating a vector of fixed size inside a class**

There are two main differences between preliminary project and coding part. Example here is from employee class but it applies to all the other classes.

1. "Fixing" vector size is done in the constructor rather than in attributes.
2. Instead of using magical constants, code uses global constant CERTIFICATES to determine how big can the vector be.



```

// BEFORE
class Employees
{
    std::vector<bool> equipmentTraining(10)

// AFTER
class Employee
{
    std::vector<bool> equipmentTraining;

// later on in constructor:
Employee::Employee(): equipmentTraining(CERTIFICATES) // global
constant set to 10 by default
{
    for(int i = 0; i < CERTIFICATES; i++)
    {
        equipmentTraining[i] = 0;
    }
}

```

Figure 5.1: Changes in how vectors sizes were defined

### Changing unsigned int to int whenever was possible

Instead of using unsigned int, I decided to go with what is considered a best practice and use int everywhere and just check whether a number is positive whenever dealing with something which is supposed to be positive like size. Example below is from good class but it applies to all the other classes.





```
// BEFORE
class Goods
{
    private:
        unsigned int weight;

// AFTER
class Good
{
    private:
        int weight

// later on in setWeight method
void Good::setWeight(int newWeight)
{

    if(newWeight > 0) weight = newWeight;
}
```

Figure 5.2: Changing unsigned int to int

### Changing setters type to void

Instead of setters having the type of whatever attribute they change and having to return this type, they are all now a void type. Example below is from good class but it applies to all the other classes.



Figure 5.3: Changing setters types to void

### 5.2.2 Good class changes

#### Methods removed

- Destructor - I decided that instead of trying to come up with the destructor on my own I will just let the compiler create default one.
- Copy constructor - As there was no real use of it I decided to remove it.

#### Methods added

No methods were added when compared to preliminary project.

### 5.2.3 Other modifications

Name of the class was changed to "Good" to show that this class is used to create single good instead of multiple goods.

### 5.2.4 Equipment class changes

#### Methods removed

- Destructor - I decided that instead of trying to come up with the destructor on my own I will just let the compiler create default one.

- Copy constructor - As there was no real use of it I decided to remove it.

#### **Methods added**

- canMoveGood sub methods - Used to divide canMoveGood method in smaller methods to simplify testing and using this method.
- occupySpaceGood - Allows us to change sizeOccupied and weightOccupied by just specifying good that is supposed to be put on the equipment.
- operator = override - Used when assigning equipment to an employee.

#### **Other modifications**

No changes were made in header file apart from changes mentioned in global section

### **5.2.5 Employee class changes**

#### **Methods removed**

- Destructor - I decided that instead of trying to come up with the destructor on my own I will just let the compiler create default one.
- Copy constructor - As there was no real use of it I decided to remove it.
- setCanRepair - Replaced with trainRepair

#### **Methods added**

- train method for Equipment, States and Hazard as opposed to train method only for equipment.
- canAssignEquipment - used in assignEquipment method to check if employee can be assigned this equipment.
- canMoveGood sub methods - added to divide canMoveGood method into smaller methods.

#### **Other modifications**

- moveGoods was changed to MoveGood as it was simpler to make and could be easily changed into moveGoods by using for loop, therefore variables inside the method were changed to one goodToMove instead of a vector and two pieces of equipment instead of vector consisting of equipments.



Figure 5.4: Changes in moveGood method

### 5.2.6 Warehouse class changes

#### Methods removed

- Destructor - I decided that instead of trying to come up with the destructor on my own I will just let the compiler create default one.
- Copy constructor - As there was no real use of it I decided to remove it.
- fireEmployee and retireEmployee - Removed because of lack of time and focusing on more important methods.
- deleteEquipment - Removed because of lack of time and focusing on more important methods.
- setSize and getSize - I don't really know why, probably forgot about those methods.

#### Methods added

- shipGoods submethods - Used to divide shipGoods method in smaller methods to simplify testing and using this method.

#### Other modifications

std::vector for allEmployees, allEquipment and allGoods consists of *pointers* to Employee, Equipment and Good objects instead of just objects

## 5.3 Practical problems

### 5.3.1 Transferring preliminary project code to coding part

First problem was the fact that code from preliminary project did not transfer well into coding part. Due to this I had to start the whole project from scratch and base on pseudocode and design ideas rather than code from preliminary project.

### 5.3.2 shipGoods method

shipGoods method was actually much more difficult to implement than I imagined and took most of my time while designing, coding and testing.

## 5.4 Positives

- It was my first object oriented project started from almost nothing and I still managed to make it work.
- Almost all the methods have tests.
- Most of the ideas in preliminary project were successfully implemented.
- Reusing methods, functions and trying to divide my program in as many functions and methods as possible.
- I was able to create all of my code using minimal number of classes which was most important to me.
- moveGoods method actually worked.
- I removed completely unnecessary feature of checking whether the good can fit on the equipment based on the volume of the good and the equipment, as just checking if its length, height and width are smaller than those from the equipment is sufficient.
- I have never before created a makefile file and was very happy after around 2 hours of work when it finally worked as intended.

## 5.5 Ideas for improvement

- NO UI, mistake done during preliminary project phase where instead of focusing on the functionality of my program I spend more time thinking about what methods I can add and more deep workings of my program.
- Instead of using *DEFINE* to create global constant I should have just used *const int CONSTANT =*

- Instead of creating those global constant in every file of my program, I should just create new file containing all the constants.
- Removing `std::cout` from tests and creating them on `std::cerr` instead
- Should have used some framework for tests and tools to check for memory leak or other undefined behavior.

## 5.6 Things I learned

- How to make makefile.
- How to translate planning phase into coding phase.
- How to make working relations between different classes.
- How to create `ifndef` guards.
- How to create object oriented programming project.
- How to use Latex.

## Chapter 6

# Ending comments

Having no experience with object oriented programming except for laboratory exercises I was able to create four different classes and relations between them so that they would actually be able to do something. Of course because that was my first time experience I made a lot of mistakes but I consider them rather easy to fix and an opportunity to improve. Overall I am very satisfied with the project I turned in.