The debugger as a learning tool for object-oriented programming

Master's Thesis January 25, 2021

John Visser

The debugger as a learning tool for object-oriented programming

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

John Visser
born in Rijswijk, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

The debugger as a learning tool for object-oriented programming

Author:	John Visser
Student id:	9894095
Email:	`J.J.Visser-TI@student.tudelft.nl`

Abstract

Novice students still have problems with the abstraction of object-oriented programming. This research shows that the debugger could be used to clarify abstraction of the object-oriented programming with the C++ program language. Our investigation spans two consecutive courses on object-oriented programming. For the introduction course on the object-oriented programming, we modified both lectures and lab sessions so that with the help of the debugger objects could be visualised, and we could walk through the program step by step. In the second course, the lectures were extended by forty-five minutes, allowing the students to do assignments during the lectures with the help of the debugger. The results of the written examination and the completed questionnaire showed that working with the debugger in both the lectures and the practical exercises had a positive influence, especially for the introductory course on object-oriented programming. Although the Corona pandemic influenced part two, the results of the written exam did improve, but it is unclear which influence the use of the debugger had during this course. Visualising objects and walking through methods step by step could help in understanding the fundamentals of object-oriented programming language C++.

Thesis Committee:

Chair:	
University supervisor:	Prof. dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	
Committee Member:	Dr. E. Demirović, Faculty EEMCS, TU Delft

Preface

During years of teaching in programming courses and in particular object-oriented programming. I became more and more convinced that the debugger tool in education can be more than just finding simple errors. The completion of my part-time Master's degree in computer science at Delft University of Technology offered me the opportunity to conduct research about using the debugger tool to clarify the concepts of object-oriented programming. This report may be interesting for those who are involved in teaching object-oriented programming.

First of all, I would like to thank Supervisor Professor Andy Zaidman for his comments and many pieces of advice during graduation, and in particular during the completion of the thesis. I would also thank colleagues for their cooperation and encouragement in completing the master's, and of course, the students. However, they were not aware that they were participating in this research.

John Visser
Delft, the Netherlands
January 25, 2021

Contents

Preface **iii**

Contents **v**

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**
- 1.1 Using the debugger . . . 1
- 1.2 The research environment. . . . 2
- 1.3 Research questions . . . 3
- 1.4 Structure . . . 4

2 Background and Related work **5**
- 2.1 The debugger tool . . . 6
- 2.2 The debugger as a clarification tool for object-oriented concepts. . . . 8

3 Experimental setup **19**
- 3.1 The methodology employed . . . 19
- 3.2 Analysis of the 2019 ”Introduction Object-Oriented programming” exam . 25
- 3.3 Analysis of the 2019 ”Object oriented programming part II” exam . . . 34
- 3.4 Adjusting the education material . . . 41
- 3.5 Conclusion regarding experimental setup . . . 54

4 Results and analysis of the experiments **55**
- 4.1 Analysis of the current ”Introduction object-oriented programming” exam . 55
- 4.2 Analysis of the current ”Object-oriented programming part II” exam . . . 62
- 4.3 Evaluation of using the debugger during the course by the students . . . 69

5 Discussion **75**

5.1 Interpretation of results 75
5.2 Threats to validity 78

6 Conclusion and Future Work **81**

Bibliography **83**

A Glossary **89**

B Exam 2019, Introduction Object-Oriented Programming (translated to English) **91**

C Exam 2020, Introduction Object-Oriented Programming (translated to English) **95**

D Exam 2019, Object-Oriented Programming, part II (translated to English) **101**

E Exam 2020, Object-Oriented Programming, part II (translated to English) **105**

F Results analysis exam OOPR1 January 2020 **111**

G Timetable of the course Introduction Object Oriented programming **113**

List of Figures

1.1 Example of inheritance. 2
1.2 Example of clarifying inheritance by using a debugger. 2

2.1 The representation of a composition and aggregration in UML. 8
2.2 Class rectangle with different types of fractions. 9
2.3 Class rectangle with fractions as primitive and as pointer variable. 9
2.4 Abstracte view of a constructor. 10
2.5 Debugging of a constructor of the class Fraction. 10
2.6 Debugging of the method `add` of he class Fraction. 11
2.7 A copy of the object from class Rectangle without problems. 11
2.8 A copy of the object from class Rectangle with pointers to the fractions. 12
2.9 A right copy of the object from class Rectangle. 12
2.10 Inheritance of the class Dog with the classes Dachshund and SaintBernard.[1] . . 13
2.11 Dog is a subset of Dachhunt and SaintBernard. 13
2.12 The order of constructor call with base constructor defined. 14
2.13 The order of constructor call without base constructor defined. 14
2.14 The order of destructor call without base destructor defined. 14
2.15 The slicing problem. 17

3.1 The first two years of the curriculum of the study programme NSE of the department HBO-ICT. 19
3.2 Distributing of the results of the exams Introduction object-orieted programming. 24
3.3 Class diagram of the exam January 2019. 27
3.4 Value of the variables, with breakpoint at line 8. 29
3.5 Visualization of the object pi of class Raspberry. 31
3.6 Copy of Raspberry without copy constructor. 32
3.7 Copy of Raspberry with copy constructor. 32
3.8 Check whether an extended board is available. 34
3.9 Class diagram of the exam March 2019. 36
3.10 Showing the `this` parameter. 37
3.11 Watching content of maps and iterators which refers to the map. 40

3.12 The base class Dog and the derived class SaintBernard 43
3.13 Steps to create an object of class SaintBernard 43
3.14 Object of class SaintBernard with a whiskey barrel 44
3.15 An example of a multiple-choice question about constructor. 44
3.16 An example of a multiple-choice question about overridden a method. 45
3.17 The lab envirionment. 45
3.18 The class diagram main code of assignment 2A. 46
3.19 Class diagram and snapshots of assignment 2A. 46
3.20 Class diagram of assignment 2B 47
3.21 The sequence diagram of assignment 2B. 47
3.22 Snapshots of assignment 2B. 48
3.23 The class diagram of assignment three. 48
3.24 During the creation of the base class. 49
3.25 Snapshots during the creation of an object of class SingleLed. 49
3.26 Snapshots of copy from a singleLed without and with copy constructor 50
3.27 A base and derived pointer refers to the same object. 50
3.28 Calling the method, which is overridden. 51
3.29 Class diagram belongs to assignment 4. 51
3.30 Code and the content of the object l1 with a vector of different types of leds. . . 52
3.31 The content of the STL container set, in Code fragment 3.19. 53

4.1 The distribution of the results of the exams Introduction object-orieted programming. 56
4.2 Class diagram of assignment 1 of the exam Introduction object-oriented programming. 58
4.3 Class diagram of assignment 2. 61
4.4 The distribution of the results of the copy constructor questions in 2017, 2019 and 2020. 62
4.5 The distribution of the results of the exams object-orieted programming part II. 64
4.6 Class diagram of the exam Object-oriented programming part II. 65
4.7 Difference in distribution between question six in 2019 en question eight in 2020. 68
4.8 Results of the m.c. questions about using the debugger with the review of previous lecture 70
4.9 Results of the m.c. questions about using the debugger during the lecture. . . . 71
4.10 Using the example code by themselves. 71
4.11 Results of question about using debugger(question 1). 72
4.12 Results of the m.c. questions during the lab sessions. 72
4.13 Debugger used with searching for errors in program(questions 10 and 11). . . . 73

5.1 The courses of the system development line. 75
5.2 The average score of the courses in the same field as object-oriented programming. 76
5.3 Some results from the PhD of Matthew Heinsen [21]. 78

B.1 Class diagram of questions A-K 91

B.2 The main code 92
B.3 Class Controller 92
B.4 My composed figure 92
B.5 Sequence diagram of method `createExtendedBoard`. 94

C.1 Class diagram of assignment 1 95
C.2 Assignment 2, Change of a battery 99

D.1 Class diagram of the exam March 2019 102

E.1 Class diagram of the exam March 2020 106

List of Tables

2.1 Implementation of a virtual method. 15

3.1 Globaal results of the exam introduction OOP. 24
3.2 Analyse of the results of exam Jan 2019. 25
3.3 Analyse of questing A. 27
3.4 Analyse of questing B. 27
3.5 Analyse of questing G. 28
3.6 Analyse of questing H. 30
3.7 Analyse of questing I. 31
3.8 Analyse of questing J. 33
3.9 Analyse of the results of exam part2 April 2019. 35
3.10 Analyse of question 3. 37
3.11 Analyse of question 5. 39
3.12 Analyse of question 6. 39

4.1 Results of the exams with all the participants in order of execution (OOPR1 is last). 56
4.2 Statistic results of between the exams OOPR1. 57
4.3 Results of the exams with participants who participated in all three examinations. 58
4.4 Result of question A. 59
4.5 Result of question B. 59
4.6 Result of question G. 59
4.7 The mean/Max per parts of question G. 60
4.8 Result of question H. 60
4.9 The mean/Max per parts of question H. 60
4.10 Result of questing I. 61
4.11 The mean/Max per parts of question I. 61
4.12 Result of question 2B. 62
4.13 Results of both OOP courses. 63
4.14 Statistic results of between the exams OOPR2. 64
4.15 Analyse of the results of exam part II April 2020. 65

4.16 Result of question four. 66
4.17 The mean/Max per parts of question four. 66
4.18 Result of question five. 67
4.19 The result per main item of question five. 67
4.20 Result of question eight. 68
4.21 The mean/Max per parts of question eight year 2020. 69

5.1 Results of the exams in the field of object-oriented programming cohort 2018. 76

F.1 Analyse of the results of exam January 2020 111

G.1 Subjects of the old OO course. 113

Chapter 1

Introduction

Nowadays, Object-Oriented Programming (OOP) is one of the most popular programming techniques that are used by software engineers [30, 19, 6]. It is used with many software development applications such as games, administrative software, embedded software [41], Android mobile application development is based on Java language [26, 42], one of the main development language of MAC OS X is objective-C [42], Microsoft Phone OS uses C# [42], and the dominant language for embedded systems programming[1] are C/C++ [49]. Although OOP has been taught at universities for years and teachers have years of experience in education in object-oriented programming, it is still difficult to understand the abstraction of object-orientated programming by novice students [57], which results in many dropouts [43]. In this report, we investigate whether an intensive way of using the debugger can be used to clarify the object-oriented programming. More specially, we focus on object-oriented programming concepts such as classes versus objects, inheritance, aggregation, constructors, destructors and other object-oriented concepts. It was investigated whether an intensive way of using the debugger can contribute to a better understanding of OO characteristics.

1.1 Using the debugger

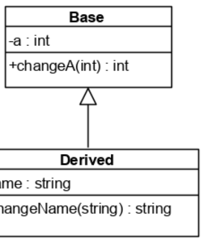
That the debugger could contribute to clarifying OOP concepts was already described by Cross & Hendrix and Barowski. This was done for the programming language JAVA and the tool JGRASP [17]. This report describes how a visual debugger (DDD)[2] [59] and the IDE of Eclipse can contribute to clarifying OOP concepts in the C++ programming language.

The advantage of the DDD debugger is that the objects can be visualised, which makes it visible that the base class is a part of the derivative class. As an example, we take the class diagram of Figure 1.1a and the corresponding Code fragment 1.1. The base class has as attribute `a` of the type integer, and the derive class has as attribute `name` of the type string. Code fragement 1.1 shows how to create the object `dr` of the derived class where

[1]website can be found at `https://www.embedded.com/2019-embedded-markets-study-reflects-emerging-technologies-continued-c-c-dominance/`

[2]website can be found at `https://www.gnu.org/software/ddd/`

the attributes a and name are assigned the values 2 and ”mr. Been”. Demonstrating that the



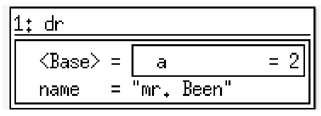
(a) Class diagram with a base and a derived class.

```
int main() {
  Derived dr(2,"mr.␣Been");
  cout<<dr.changeName("Roy")<<endl;
  return 0;
}
```

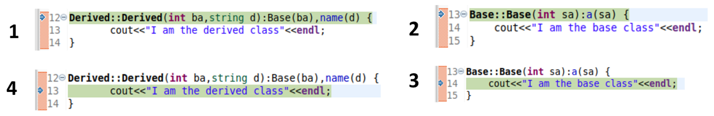
Code fragment (1.1) Creation of a derived class.

Figure 1.1: Example of inheritance.

base class is actually part of the derived class could be clarified by visualising the object with the DDD debugger, as shown in Figure 1.2a. In this case, the object dr is displayed as a box containing the attribute name and the base class with attribute a. With the name of the base class between <>. When an object of a derived class is created, the debugger can



(a) An object of a derived class by the DDD debugger.



```
Derived::Derived(int ba,string d):Base(ba),name(d) {
        cout<<"I am the derived class"<<endl;
}
```

```
Base::Base(int sa):a(sa) {
    cout<<"I am the base class"<<endl;
}
```

(b) creation of a derived object with the Eclipse debugger (GDB).

Figure 1.2: Example of clarifying inheritance by using a debugger.

also be used to clarify that the constructor of both the derived and the base class are called. This is made visible in Figure 1.2b where the numbers indicate the sequence of events. This example shows that visualisation of an object could help to clarify the principle that a base class is a part of the derived class.

1.2 The research environment.

The experiment was done at the section Network System Engineering of the department HBO-ICT which is a part of Faculty of I T & Design of the Hague University of Applied Sciences location Delft. This section is teaching two consecutive object-oriented programming courses. The first course treats the basic concepts of object-oriented programming, the second course delves deeper into the OOP. Further information on the content of the courses is described in Chapter 3.1.1. Each course consists of a theoretical and a practical part and finished with a written exam. Furthermore, the students had to demonstrate their programmes to the instructor who asked questions about it. During lab sessions, students can use different tools in order to develop their programming skills. One of the tools they can use in both practicals of the OOP courses, is the debugger to locate bugs.

To be able to perform the experiment, both courses have been adapted. During the first course, the student was taken more by the hand, so the student had to use the debugger. It was verified whether the student had actually used the debugger during the practical course. In the subsequent course, the emphasis was more on clarifying the concepts during the lectures and letting the students practice with them.

The first course, Introduction object-oriented programming both the lectures and the practical lessons have been adapted. In the subsequent course (Object-oriented programming part II) only the lectures have been adapted, whereby the lectures have been extended from 45 to 90 minutes so that there has been sufficient space for the students to make short assignments during the lectures and applying the debugger. The analysis was done based on both written exams and based on a questionnaire completed by the students in the last lecture of the course Introduction object-oriented programming.

1.3 Research questions

To investigate whether the debugger could be used as a learning tool during object-oriented programming lessons, one hypothesis was formulated, which is described in Section 1.3.1. Several secondary questions, described in Section 1.3.2, were formulated to indicate the specific directions of the research.

1.3.1 The hypothesis

The hypothesis of this thesis about using the debugger as an auxiliary tool during object-oriented programming lessons is:

People who use a debugger when learning object-oriented programming better understand the concepts.

By changing both the lectures and the lab sessions so that the debugger played a prominent rule, it was possible to obtain an indication whether the more intensive use of the debugger could contribute to an improved understanding of object-oriented programming by novice students.

1.3.2 Research Questions

To support the hypothesis of Section 1.3.1 , the research questions below have been formulated. The questions related to the use of the debugger and which object-oriented problems could be solved with the debugger. By answering these questions, the main research question could also be answered.

Research has shown that working with a debugger is difficult for beginning students [23, 2] and even for experienced programmers do not always use the debugger and often work with printf statement [7]. It is therefore very likely that in previous courses related to programming, several students have not worked with the debugger, or have worked sporadically. Because this research aims to clarify concepts of the object-oriented programmer with the help of the debugger, it is essential that students have already been introduced to a debugger so that the focus could be more on the object-oriented programming instead of working with the debugger. Hence, the first request question (RQ) deals with the students' skills in operating a debugger.

RQ1: How can we ensure that students have sufficient skills to use a debugger during the introduction course C++?

By using the debugger as a learning tool, it was intended that each student would also use the debugger during the practical lessons so that the object-orientated feature could be explained.

RQ2: How to ensure that the debugger will be used during the lab sessions?

The concept of object-oriented programming is extensive, and there is too little time to deal with every specific detail in depth; moreover, it is desirable that the students keep their attention. In order to use the debugger as an additional tool during the lessons, it is essential to know which object-oriented features the students find difficult.

RQ3: What are the problems that students encounter while trying to understand OO concepts? (whether or not using a debugger)

After it became known which problems the students had with object-oriented programming, it was essential to know which of these problems could be clarified with the help of a debugger.

RQ4: Which are the concepts of object-oriented programming that can be clarified by using a debugger?

1.4 Structure

This thesis begins by describing the background information about recent works to clarify the object-oriented concepts and how could be used the debugger to clarify the object-oriented concepts in Chapter 2. Chapter 3 describes the experimental setup, with the methodology used and the environment which the experiment took place. The analysis of the previous exams of the courses Introduction object-oriented programming and Object-oriented programming part II. Furthermore, adjusting the education material. Chapter 4 describes the analysis of the written exam of both object-oriented programming courses and the results of the students' evaluation of the course Introduction object-oriented programming. Chapters 5 describes the discussion. The conclusion and future work are discribed in Chapter 6.

Chapter 2

Background and Related work

Object-Oriented Programming (OOP) plays a significant role in modern software development [52]. Android mobile application development is based on the Java codes [26, 42], one of the main development language of MAC OS X is objective-C [42], Microsoft Phone OS uses C# [42] and C/C++ are the dominant languages for embedded systems programming [49]. It is therefore important that object-oriented programming is taught.

At a lot of universities are teaching object-oriented programming [57, 54]. Because of learning the object-oriented approach is difficult for the most novice students [27, 36, 25, 17, 48, 7, 55], there is a lot of research about this topic. For example Hubwieser and Berges did an experiment in which students without any programming background are able to learn a few global principles of object-oriented programming with as little (human) instruction as possible in two and half days [29]. The result was quit satisfactory: most students were surprised that they were able to learn 'programming' in two and half days, but it is unclear how much help they got from the tutors. There are also educative languages, such as VLT-OOP [52], scratch developed by MIT [37], Alice [3] developed by Carnegie Mellon University, but Ezenwoye [22] concluded that educational programming languages have seen very little use in academia. In general there are two different insights how to apply the programming knowledge to the students, objects-first or objects-late methods.

The main difference between both aspects is: should the students learn the principles of OO (class, objects, methods,..) first or should the students learn the principles of imperative programming (loop statements, conditional statements,..) first. Uysal did a research about the effects of objects-first and objects-late methods [55] and one of their conclusion was that students who did objects-first could easily understand the concepts of class and object relationship instead of concentrating on structural programming facts but with the objects-first method were more visualisation and instructional activities. During this research students worked with the tool BlueJ[1]; this is a pedagogical tool to support objects-first [14, 12]; it is unclear what the influence of the tool BlueJ is during this research.

Sometimes there is a contradiction between different studies about understanding object-oriented programming. Georgantaki and Retalis described how the tool BlueJ the students helped to understand OOP [25], but a few years after Georgantaki and Retalis had done their

[1] website can be found at `https://bluej.org/`

experiment, Bennedsen and Schulte did an experiment with BlueJ [8] too. They concluded that the students, who used BlueJ did not perform better. There are different insights and tools for learning the object-oriented development software according to the object-oriented paradigm. But there is a tool that can be used independently of the object-oriented learning insights to clarify the object-oriented paradigm, and that is the debugger.

In addition to helping to detect errors in program code, features of the debugger can also be used to clarify the object-oriented principle. An example is the tracer, which can be used to execute each statement separately. This feature makes it possible to clarify when creating an object the constructor is called. Already in 2002 described Cross, Hendrix and Barows that the use of a debugger had a positive effect on program comprehension [17]. A problem with using the debugger is that students and professional programmers often avoid the debugger. Many professional programmers and students find working with a debugger difficult and prefer to use the traditional ”printf” debugging [7]. The question with using the debugger was: ”how to find a method that students would have to use the debugger”.

This report examines whether another way and more intensive use of a debugger can lead to a better understanding of object-oriented programming. In the next session is described, how and which debugger tools were used during this research.

2.1 The debugger tool

Debuggers are important tools to find bugs in a program. The use of a debugger is not used as often as expected, whereas a lot of people still use the traditional ”printf” debugging style [7]. There has been a various developments in debugger tools over the last years such as back-in-time debuggers [53], object-centric debuggers [50], where breakpoints are references to a particular object instead of a reference to source code, there are also domain-specific debuggers [15], where you can debug at a higher abstraction level (from application domain). A debugger that is specialized in trace-oriented metaprogramming language [35]. There are also several advanced on-line debugging techniques [20].

To search bugs in a running program while using a debugger, with the tracer can be walked step by step through the program. This can be easily done in modern IDE such as Eclipse and Visual Studio. Because of the facts that by using a modern debugger a program can easily decompose, it could be helpful to understand some aspects of object-oriented programming. If an object is constructed, firstly you see how the constructor of the derived class will be invoked, secondly how the constructor of the base class will be invoked. As example JavalinaCode is designed for teaching object-oriented programming in Java. It aims to enhance student programmers’ programming skill and to help them understand object-oriented design concepts [58].

The use of a debugger has been found as difficult by students and especially with novice students. It is a lot easier to use a ”printf” statement instead of using the (symbolic) debugger at the beginning, but recently research [32] shows that in more complex projects 75% of the students use a symbolic debugger.

Researching with students who had to use a symbolic debugger so that they could easily understand object-oriented programming can lead to some challenges. First, the IDE and debugger had to be stable and easier to use. Second, the IDE and debugger had to be cross-platform (most students use windows, but some students use mac or Linux). Third, the object-oriented program language that would be used is C++; that is because the department where this research was taken place, is taught in embedded programming. Fourth, the debugger had to be able to debug remotely because the students have to learn to use an embedded platform (the lab sessions takes place mainly on a Raspberry). Fifth, the debugger had to be able to visualise objects. sixth, the debugger had to be available free of charge. Finally, students had to be able to prove that they had used the debugger.

2.1.1 Choice of the debugger

This research aimed to use the debugger to make the object-oriented programming more comprehensible to novice students rather than to detect errors in (complex) programmes. Furthermore, it was desirable that the students already had some experience in using a debugger. Because the students had already worked with a symbolic debugger in previous courses, the symbolic debugger was used during this research.

There are many IDEs for developing C++ programmes, some of them are: DEV C++, Code Blocks, QT creator, Eclipse, Visual Studio, Visual Code, Netbeans, etc. Some of the most used IDE's are Eclipse and Visual Studio [7]. In order to clarify the object-oriented programming, it is desirable that objects and the relationships between the objects are represented visually. It was also essential to be able to work remotely with the debugger.

None of the above IDEs was able to visualise objects in a simple way. Because the visualisation of objects is a crucial point to clarify object-oriented programming in this research, the Data Display Debugger (DDD) was chosen. The DDD debugger is a graphical user interface to the GDB debugger [59]. The advantage of the DDD debugger are:

- Objects and relationships between objects can be visualised.
- Easy to install in a linux environment (apt-get install ddd).

The disadvantages of the DDD debugger are:

- It is not an IDE, just a debugger.
- Cannot debug remotely (a VNC viewer or xserver is needed).
- The tracer displays the code less clearly.

To be able to debug remotely and for a clear display of the tracer, we have decided to also work with the Eclipse IDE. The advantages of the Eclipse debugger are:

- The debugger works well and reliably.
- Remote debugging is possible.
- It is a cross plattform (C++ for the MAC is officially not supported for Visual Studio by microsoft[2]).

[2]website can be found at `https://visualstudio.microsoft.com/vs/mac/`

- Eclipse platform is widely used, especially for the development of embedded platforms[3].

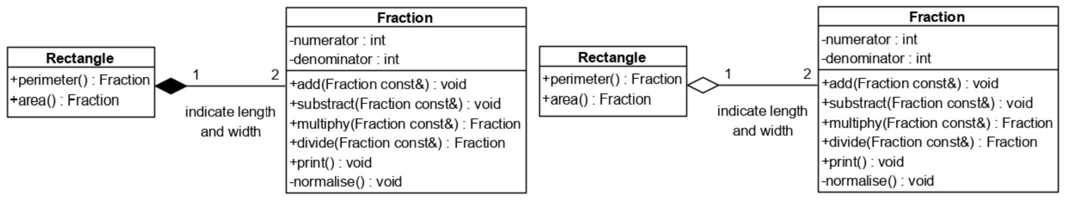
Two debuggers were used, one to visualise the objects and the other to walk through the program with the tracer and to display the variable structurally. The next session will discuss which object-oriented concepts can be clarified by the debugger.

2.2 The debugger as a clarification tool for object-oriented concepts.

Object-oriented programming has several specific concepts that are difficult to understand by novice students. This section describes how some of these concepts can be clarified with the help of the debugger. These concepts are mainly used during the introductory course object-oriented programming, but also during the object-oriented programming course part II. First, the distinction between a composition and an association in the C++ programming language is discussed. Second, the creation of an object. Third inheritance in C++. Fourth, operator overloading. Finally, the STL vector.

2.2.1 Difference between composition and association

The Unified Modeling Language (UML) has many types of relationships [5]. For example, Figure 2.1a shows a composition of a square with two fractures, and Figure 2.1b shows an aggregation of it. The difference between them is that a composition has a much more fixed relationship than aggregation. In the C++ programming language, a distinction could make between these two.


Rectangle
+perimeter() : Fraction
+area() : Fraction
1
2
indicate length and width
Fraction
-numerator : int
-denominator : int
+add(Fraction const&) : void
+substract(Fraction const&) : void
+multiphy(Fraction const&) : Fraction
+divide(Fraction const&) : Fraction
+print() : void
-normalise() : void
Rectangle
+perimeter() : Fraction
+area() : Fraction
1
2
indicate length and width
Fraction
-numerator : int
-denominator : int
+add(Fraction const&) : void
+substract(Fraction const&) : void
+multiphy(Fraction const&) : Fraction
+divide(Fraction const&) : Fraction
+print() : void
-normalise() : void

(a) A composition in UML. (b) An aggregration in UML.

Figure 2.1: The representation of a composition and aggregration in UML.

In contrast to JAVA and C#, it is possible to use an object as a primitive variable in C++. Figure 2.2 shows how objects of the class Fraction can be approached by the class Rectangle. This can be done as primitive variable, with a pointer and as a reference. The composition of Figure 2.1a can be implemented by means of primitive variables as shown in Figure 2.2a, because if the rectangle is deleted, both fractions are also deleted.

If an object of the class Rectangle, which is implemented in Figure 2.2a, is created, both fractions are created too. If an object of the class RectanglePT, wich is implemented in

[3]website can be found at `https://en.wikipedia.org/wiki/List_of_Eclipsebased_software`

```
class Rectangle {

   .........
   .........

  private:
     Fraction length;
     Fraction width;

};
```

(a) Object as primitive.

```
class RectanglePt {

   .........
   .........

  private:
     Fraction *length;
     Fraction *width;

};
```

(b) Pointer to object.

```
class RectangleRef {

   .........
   .........

  private:
     Fraction &length;
     Fraction &width;

};
```

(c) Object as reference.

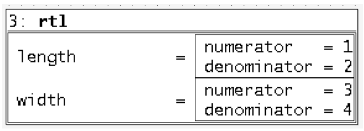
Figure 2.2: Class rectangle with different types of fractions.

Figure 2.2b, is created, both fractions are not directly created. It is depend of the constructor, if the Fractions will be created. If an object of a rectangleRef, which is implemented in Figure 2.2c, is created, both fractions are not directly created but they have to get a value. The compiler checks this, and gives an error if the fraction does not get a value. If an object of the class Rectangle Figure 2.2a is deleted, both fractions are deleted too. If an object of the class RectanglePt Figure 2.2b is deleted, it is not automatic that the fractions will be deleted. It depends on the implementation of the destructor. If an object of the class RectangleRef Figure 2.2c is deleted, both fractions will not be deleted.

The difference between implementations of the class Rectangle, RectanglePt and RectangleRef can be given more clearly with a visual debugger such as DDD, which shows the objects of Code fragment 2.1 in Figure 2.3.

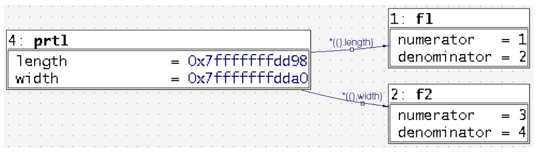
```
Fraction f1(1,2);
Fraction f2(3,4);
Rectangle    rt1(f1,f2);
RectanglePt  prt1(f1,f2);
RectangleRef ref1(f1,f2);
```

Code fragment 2.1: implementation of different types of relationships.



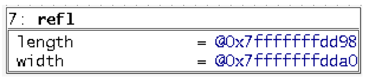
```
3: rt1
length    =  numerator   = 1
             denominator = 2
width     =  numerator   = 3
             denominator = 4
```

(a) A composition.



```
4: prt1
length   = 0x7fffffffdd98
width    = 0x7fffffffdda0

1: f1
numerator   = 1
denominator = 2

2: f2
numerator   = 3
denominator = 4
```

(b) An aggregration with pointers.



```
7: ref1
length    = @0x7fffffffdd98
width     = @0x7fffffffdda0
```

(c) An aggregration with references.

Figure 2.3: Class rectangle with fractions as primitive and as pointer variable.

Figure 2.3a shows the fixed relationship of a composition between a rectangle and both fractions, while Figure 2.3b shows the looser relationship of an aggregation. Figure 2.3c shows the implementation using references. This is done by putting @ in front of the address. A reference in C++ indicates a different name for an object.

2.2.2 Creating an object

The constructor is one of the most important parts of a class and is called if an object is created. In fact, the constructor looks like a method with the following features.

- A constructor has the same name as the class.
- A constructor has no return type, not even void.

An abstracte view of a constructor is shown in Figure 2.4. In the construction initialisation list [44, 38], that is between the colon and the curly brace, are each attribute of the class is initialised. If the constructor is called, firstly the initialisation list is carried out and finally the body is executed.

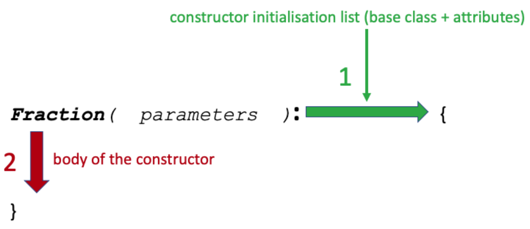

constructor initialisation list (base class + attributes)
1
Fraction(parameters): {
2
body of the constructor
}

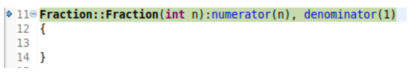
Figure 2.4: Abstracte view of a constructor.

Calling the constructor can be clarified with the debugger by putting a breakpoint on line 1 of Code fragment 2.2 and jumping to the constructor with the "step in" function of the debugger.

```
1  Fraction f1(3);
2  Fraction f2(2,4);
3  f2.add(4);
```

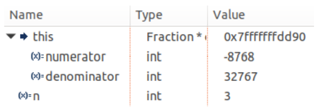
Code fragment 2.2: Creation of objects with different constructors.

After performing this step with the Eclipse debugger, Figure 2.5a is obtained. It is visible



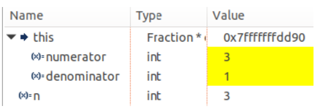
```
Fraction::Fraction(int n):numerator(n), denominator(1)
{

}
```

(a) The constructor of the class Fraction.



```
Name              Type        Value
▾ ➜ this          Fraction *  0x7fffffffdd90
    (x)= numerator    int     -8768
    (x)= denominator  int     32767
  (x)= n          int         3
```

(b) The attibutes before initilisation.



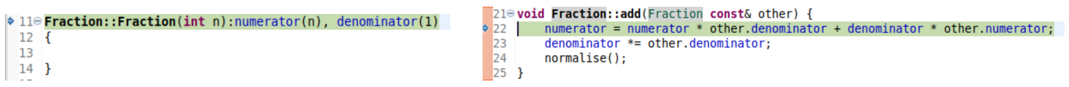
```
Name              Type        Value
▾ ➜ this          Fraction *  0x7fffffffdd90
    (x)= numerator    int     3
    (x)= denominator  int     1
  (x)= n          int         3
```

(c) The attibutes after initilisation.

Figure 2.5: Debugging of a constructor of the class Fraction.

that the debugger jumps to the constructor with one parameter and that the attributes are not yet initialised (Figure 2.5b). Because the attributes do not have a default value, the denominator must also be set to the value one. After performing the next step, it becomes visible that the attributes are initialised, as is shown in Figure 2.5c. So, the debugger can be used to show that the relevant constructor is being called and that the attributes should be initialised.

The constructor is also be called, if the compiler performs a casting. This is done, if line 3 of Code fragment 2.2 is executed. The add method of the class Fraction is defined as follows: `void add(Fraction const&);`. The compiler has arranged for the casting from an int to a Fraction. The fact that the compiler performs a casting can be clarified with the debugger. This can be done if line 3 of Code fragment 2.2 is executed by the "step in" function of the debugger. First a fraction is created (Figure 2.6a) and finally, the method



```
Fraction::Fraction(int n):numerator(n), denominator(1)
{

}
```

```
void Fraction::add(Fraction const& other) {
    numerator = numerator * other.denominator + denominator * other.numerator;
    denominator *= other.denominator;
    normalise();
}
```

(a) The constructor of the class Fraction. (b) The add method of the class Fraction.

Figure 2.6: Debugging of the method `add` of he class Fraction.

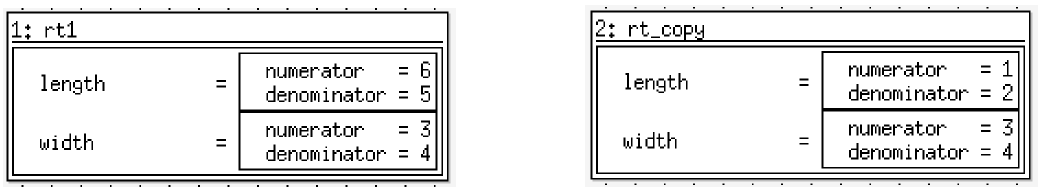
add is executed (Figure 2.6b). If an object is created directly or indirectly, the constructor is called. There is one exception when an object is copied; the copy constructor is called.

The copy constructor. Because the programming language C++ has a default copy constructor, contrary to programming languages such as JAVA and C# [16, 45], it is possible that the desired result is not achieved. The copy constructor is called when an object is copied. By default, the default copy constructor is called, which means that all the attributes of the object are copied, which in most cases does not give a problem. However, sometimes it will give a problem, mostly when a copy is done with pointers to objects.

```
1   Fraction f1(1,2);
2   Fraction f2(3,4);
3   Rectangle rt1(f1,f2);
4   Rectangle rt_copy(rt1);
5   rt1.changeLength(6,5);
```

Code fragment 2.3: Copy of an object without problems.

After copying the rectangle on line 4 of Code fragment 2.3, another value is assigned to the attribute `length` of object `rt1` on line 5. As Figure 2.7 shows, this does not cause any problems. The attributes keep its own values, object `rt1` has its change value, which is



```
1: rt1
length    =   numerator   = 6
              denominator = 5
width     =   numerator   = 3
              denominator = 4
```

```
2: rt_copy
length    =   numerator   = 1
              denominator = 2
width     =   numerator   = 3
              denominator = 4
```

(a) The orginal rectangle. (b) The copy rectangle.

Figure 2.7: A copy of the object from class Rectangle without problems.

shown in Figure 2.7a, and object `rt_copy` keeps its origin value, which is shown in Figure 2.7b. The default copy constructor works well.

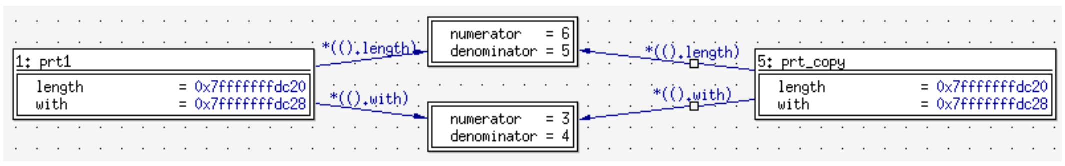
When the rectanglePt of Figure 2.2b is copied on line 4 of Code fragment 2.4, the pointers to the fractions are copied and not the fractions themselves. If the attribute `length` of object `prt1` changes on line 5 of Code fragment 2.4, the attribute `length` of object `prt_copy` also changes. That happens, because they are the same fractions.

```
1   Fraction f1(1,2);
2   Fraction f2(3,4);
3   RectanglePt prt1(f1,f2);
```

```
RectanglePt prt_copy(prt1);
prt1.changeLength(6,5);
```

Code fragment 2.4: Copy of an object with problems.

After visualisation of the objects by the DDD debugger, Figure 2.8 is displayed, which can provide an enlightening insight to the students.



```
1: prt1
  length  = 0x7fffffffdc20
  with    = 0x7fffffffdc28

numerator   = 6
denominator = 5

numerator   = 3
denominator = 4

5: prt_copy
  length  = 0x7fffffffdc20
  with    = 0x7fffffffdc28
```

Figure 2.8: A copy of the object from class Rectangle with pointers to the fractions.

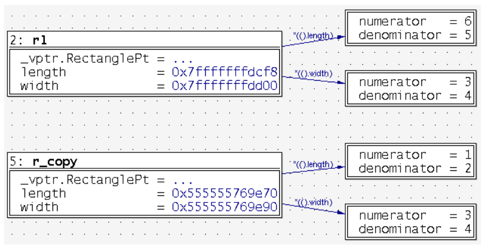
If it is desirable that the fractions will be copied too, the copy constructor has to be overloaded. The prototype of the copy constructor of the class RectanglePt is `RectanglePt(const RectanglePt&);` The implementation of the copy constructor can be done as follows:

```
RectanglePt::RectanglePt(const RectanglePt& p) {
      length = new Fraction(*p.length);
      with=new Fraction(*p.with);
}
```

The understanding of the existence of the copy construction can be clarified with the debugger. Place a breakpoint at the statement **RectanglePt r_copy(r1);** and proceeding step by step, the debugger will jump to the copy constructor. The result of executing the copy constructor is shown in Figure 2.9.

By using the debugger to explain the copy constructor, it might be more evident that:

1. The existence of a copy constructor.
2. The problems with pointers as attribute.
3. There is direct access to the private attribute of the object with the same type.



```
2: r1
  _vptr.RectanglePt = ...
  length            = 0x7fffffffdcf8
  width             = 0x7fffffffdd00

numerator   = 6
denominator = 5

numerator   = 3
denominator = 4

5: r_copy
  _vptr.RectanglePt = ...
  length            = 0x555555769e70
  width             = 0x555555769e90

numerator   = 1
denominator = 2

numerator   = 3
denominator = 4
```

Figure 2.9: A right copy of the object from class Rectangle.

Off course, with this solution, the destructor has to know that these fractions have to be deleted.

2.2.3 Inheritance

The principle of inheritance is often difficultly to understand for novice students. There is a base class and a derived class. The base class is a subset of the the derived class and a method can be overloaded or overridden.

In the example which is shown in Figure 2.10 the base class is the class **Dog** and the derived classes are the classes **SaintBernard** and **Dachhund**. Both derived classes are dogs. The difference between the class **Dog** and the derived classes is the sound that their make. The dachhund barks ”kef kef” and the Saint Bernard barks ”woef woef”. In addition the Saint Bernard also has a whisky barrel.

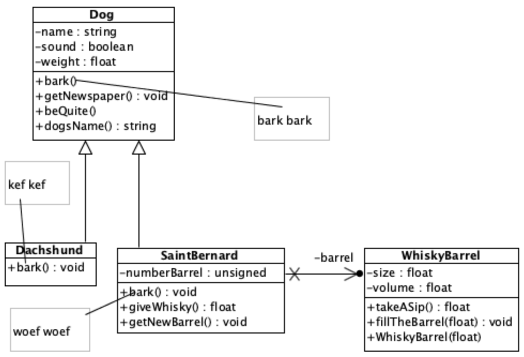


Figure 2.10: Inheritance of the class Dog with the classes Dachshund and SaintBernard.[4]

A number features of the inheritance principle can be clarified with the debugger. Firstly, the principle that the base class is a subset of the derived class. Secondly, the difference between method overloading and overriding.

Base class is a subset of the derived class. In the case of inheritance of a class, the base class is embedded in the derived class [38]. This embedding can be visualised by the DDD debugger, which could have a clarifying effect. If Code fragment 2.5 is executed and a breakpoint is set on line 5, the variables `barrel`, `sb1` and `dh1` can be visualised. The result is shown in Figure 2.11.

```
1  WhiskyBarrel barrel(10);
2  barrel.fillTheBarrel(6);
3  SaintBernard sb1(150,"Sergeant␣Dog",\&barrel);
4  Dachshund dh1(125,"Danke");
5  dh1.bark();
```

Code fragment 2.5: Creation the derived classes SaintBernard and Dashhund.

It is clarifying to see that the base class **Dog** is a subset of the derived classes **Dachhund** and **SaintBernard**. Especially object `sb1`, it is clear that dog with the attributes `name` and `weight` is a subset of SaintBernard which also has attributes `numberBarrel` and a pointer to `barrel`.

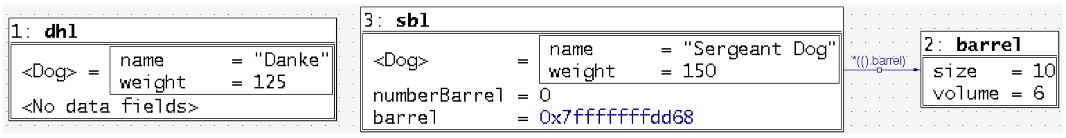
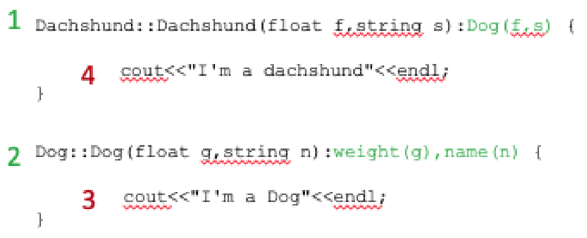

```
1: dh1
<Dog> = name     = "Danke"
        weight   = 125
<No data fields>

3: sb1
<Dog>          =  name     = "Sergeant Dog"
                  weight   = 150
numberBarrel   = 0
barrel         = 0x7fffffffdd68

2: barrel
size   = 10
volume = 6
```

Figure 2.11: Dog is a subset of Dachhunt and SaintBernard.

During the creation of objects `dh1`, which can be followed by a tracer of a debugger, both Dachshund and Dog constructors are called. The order of calling constructors is shown in Figure 2.12.

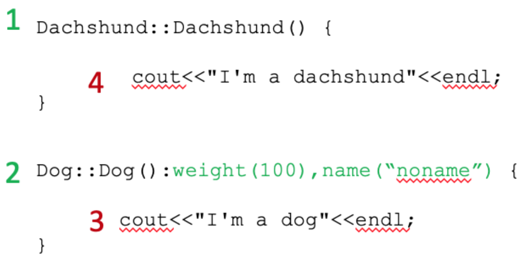
[4]The idea of colleague Harry Broeders.

Firstly, the constructor initialisation list of the class **Dachshund** is called (1) where the base class constructor **Dog** is defined in the list. Secondly, the derived constructor of the class **Dog** is called (2) where the attributes weight and name are initialised. Thirdly, the body of the constructor **Dog** is executed (3). Finally, the body of the constructor **Dachshund** is executed (4).



```
1 Dachshund::Dachshund(float f,string s):Dog(f,s) {
      4 cout<<"I'm a dachshund"<<endl;
  }

2 Dog::Dog(float g,string n):weight(g),name(n) {
      3 cout<<"I'm a Dog"<<endl;
  }
```

Figure 2.12: The order of constructor call with base constructor defined.

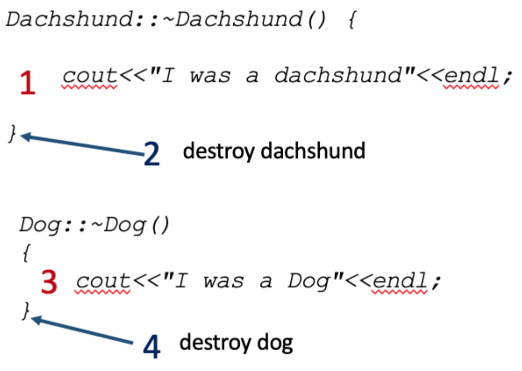


```
1 Dachshund::Dachshund() {
      4 cout<<"I'm a dachshund"<<endl;
  }

2 Dog::Dog():weight(100),name("noname") {
      3 cout<<"I'm a dog"<<endl;
  }
```

Figure 2.13: The order of constructor call without base constructor defined.

If no base class constructor is defined in the initialisation list of the derived class, the default constructor (constructor without parameters) of the base class is called. Figure 2.13 shows the sequence when creating an object if there are no parameters. By walking through the constructor step by step with the tracer, it can be clarified that the base class is created even if there are no parameters.

If an object of a derived class is destroyed, the destructor of the derived class is called. Firstly, the body of the destructor of the derived class will be called then the attributes of the derived class. Secondly, the body of the destructor of the base class will be called then the attributes of the base class. Figure 2.14 shows the destroying of an object from the class **Dachshund**. Firstly, the body of the destructor from the class **Dachshund** is executed. Secondly, dachshund itself is destroyed. Thirdly, the body of the destructor from the class **Dog** is executed. Finally, dog itself is destroyed.



```
Dachshund::~Dachshund() {
  1 cout<<"I was a dachshund"<<endl;
}

 Dog::~Dog()
 {
  3 cout<<"I was a Dog"<<endl;
 }
```

Figure 2.14: The order of destructor call without base destructor defined.

Difference between method overloading and overriding. One of the features of OO is method overloading and especially method overriding. Method overloading means that methods have the same name but with different types [51, 38]. Methods can also be overridden, this means that a method of the derived class can also be called, by a method of the base class which has the same name and type. This can only be done in C++ with references and pointers. This is dynamic binding and can be done to use the ”`virtual`” keyword. The `virtual` keyword is used in front of a method of the base class with declaring [38]. The method `void bark() const;` of the base class Dog will be `virtual void bark() const;`. The method `void bark() const;` of the derived class can remain unchanged but it will have been recommended to write the keyword ”*override*” behind the method `void bark() const override`; of the derived class since C++11 [38]. The compiler gives an error if the signature of the base class changes and not the derived class.

To explain polymorphism and the difference between method overloading and overriding, the method *`void bark() const;`* of Figure 2.10 is used as example. The implementations of diverse methods of bark are shown in Table 2.1. Every class has its own

Class	**Implementation**
Dog	`void Dog::bark() const{ cout<<"bark bark"<<endl; }`
Dachshund	`void Dachshund::bark() const { cout<<"kef kef"<<endl; }`
SaintBernard	`void SaintBernard::bark() const { cout<<"woef woef"<<endl; }`

Table 2.1: Implementation of a virtual method.

implementation of the method `void bark() const;`, so the method bark of the class **Dog** print ”bark bark”, the class **Dashhund** ”kef kef” and the class **SaintrBernard** ”woef woef”. To show method overriding, Code fragment 2.6 is used. The code can be split in 3 parts:

1. Initialisation and creation of objects from the classes **Dog**, **Dachshund** and **SaintBernard** (line 1 till 5).
2. Execution of the method `void bark() const;` of all the objects (line 7 till 9).
3. Using the references to objects of the classes **Dachshund** and **SaintBernard** and run the method `void bark() const;` of both objects (line 11 till 14).

```
1  WhiskyBarrel barrel(10);
2  barrel.fillTheBarrel(6);
3  SaintBernard sb1(150,"Sergeant␣Dog",&barrel);
4  Dachshund dh1(125,"Danke");
5  Dog dg(75,"Pluto");
6
7  dg.bark();
8  dh1.bark();
9  sb1.bark();
10
11 Dog& refdog=dg;
12 Dog& refBernard=sb1;
13 refdog.bark();
```

```
14  refBernard.bark();
```

Code fragment 2.6: Difference between overloading and overriding.

By jumping into the method `bark` with the tracer, it is possible to see which method is being called. Do we jump form line 7, 8 and 9, then we jump into the `bark` method of the relevant type. But by jumping from line 13 into the method `bark`, the method `bark` of the derived class is called.

Virtual destructor. A destructor should be declared virtual if the class has at least one virtual method. That is because if an object is manipulated through the interface provided by a base class, it is also often deleted through that interface [51]. With the help of the debugger, the difference between a virtual and a non-virtual destructor can be demonstrated by using the tracer and walking from line 5 of Code fragment 2.7 step by step.

```
1  WhiskyBarrel barrel(10);
2  barrel.fillTheBarrel(6);
3  Dog* d= new  SaintBernard(150,"Sergeant␣Dog", &barrel);
4  d->bark();
5  delete d;
```

Code fragment 2.7: An example of the Slicing problem.

If there is a reference or pointer to a base class, without a virtual destructor only the destructor of the base class is called. With a virtual destructor both the base and the derived destructor are called.

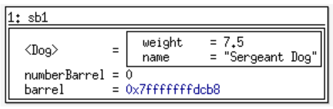
Slicing problem. With the help of the debugger, the slicing problem can be clarified. The slicing problem means that a derived class is assigned to a base class. The problem that arises is that information is lost. Demonstrating that information is lost can be done by a combination of making objects visible and using the tracer.

On line 3 of Code fragment 2.8, the derivative object `sb1` of the class **SaintBernard** (Figure 2.10) is created which is assigned to the reference `d` of the class **Dog** (line 5) and to the variable `d2` of the class **Dog** (line 6).

```
1  WhiskyBarrel keg(1.1);
2  keg.fillTheBarrel(0.5);
3  SaintBernard sb1("Sergeant␣Dog",7.5,&keg);
4
5  Dog &d(sb1);
6  Dog d2(sb1);
7  d.bark();
8  d2.bark();
```

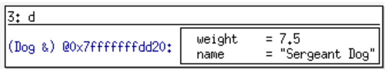
Code fragment 2.8: An example of the Slicing problem.

By visualising the object `sb1`, a clear distinction is made between the attributes `weight` and `name` of the base class and the attributes `numberBarrel` and `barrel` of the derived class, as shown in Figure 2.15a. By assigning a reference of the base class to the object `sb1` (line 5) and assigning the object `sb1` to a variable of the class **Dog** (line 6), the derived attributes are no longer visible by visualising them, but in the case of object `d` (Figure 2.15b) they are



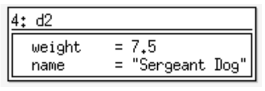
```
1: sb1
<Dog>        =  weight   = 7.5
                name     = "Sergeant Dog"
numberBarrel = 0
barrel       = 0x7fffffffdcb8
```

(a) Object sb1.



```
3: d
(Dog &) @0x7fffffffdd20:  weight   = 7.5
                          name     = "Sergeant Dog"
```

(b) Reference d to object sb1.



```
4: d2
weight   = 7.5
name     = "Sergeant Dog"
```

(c) Object d2.

Figure 2.15: The slicing problem.

still present in contrast to object `d2` (Figure 2.15c). Figure 2.15b shows the address of this object, which indicates that it is a reference. By using the 'step in' function of the tracer, it can be clearly demonstrated that on line 7 the function `bark` of the derived class is called and on line 8 the function `bark` of the base class is called. This means that the derivative part of variable `d2` does not exist (is cut off). By making the objects more comprehensible in combination with calling a virtual function, the slicing problem could be clarified.

Chapter 3

Experimental setup

3.1 The methodology employed

The participants of this research were second year students of Hague University of applied science of the academic IT & Design department HBO-ICT study programme Network System Engineering in Delft. They followed the courses Object-Oriented Programming (OOP) which are second-year courses. Figure 3.1 shows how the curriculum was during this research. The two object-oriented programming courses are dark blue and the courses in the same field blue. When the course Introduction OOP started, the students should have knowledge about imperative and microcontroller programming that has been taught in the foundation year and object-oriented modelling that was taught in the term before introduction OOP. At the same time, when the course introduction OOP was taught, the home

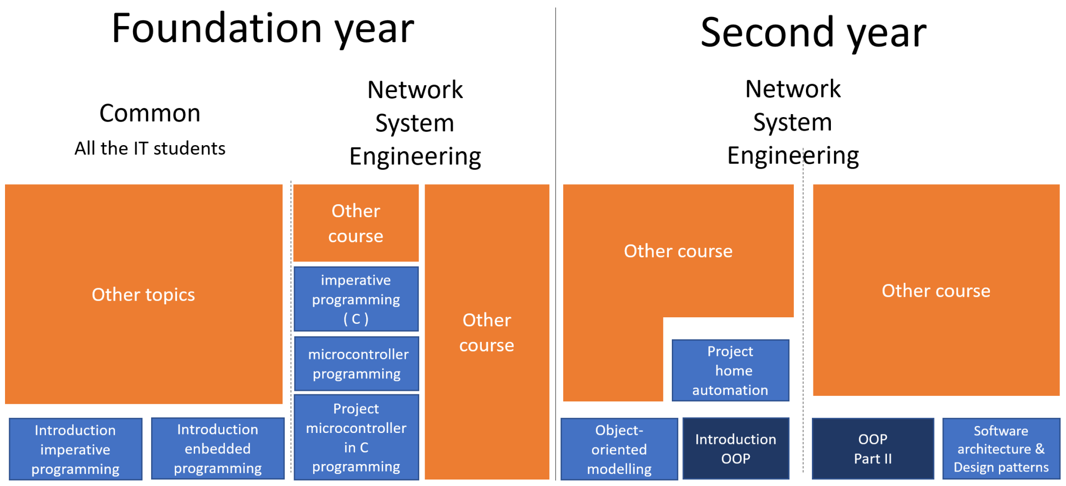

Foundation year
Common
All the IT students
Network System Engineering
Other topics
Introduction imperative programming
Introduction enbedded programming
Other course
imperative programming (C)
microcontroller programming
Project microcontroller in C programming
Other course
Second year
Network System Engineering
Other course
Project home automation
Object-oriented modelling
Introduction OOP
Other course
OOP Part II
Software architecture & Design patterns

Figure 3.1: The first two years of the curriculum of the study programme NSE of the department HBO-ICT.

automation project was also taught. After the course OOP part II, the course Software architecture & design patterns was taught. Both the OOP courses and the related courses of the study programme Network System Engineering except the projects consist of a theoretical and a practical part.

3.1.1 Explanation of the courses

The experiment with the debugger as a learning tool applied to both OOP courses. The learning objectives of both courses are:

- Introduction object-oriented programming.
  1. The student can: from a UML diagram, implement C++ code and vice versa.
     - Implementing a class.
     - Implementing an association, aggregation and composition.
     - Calling a method from a sequence diagram.
     - From code to a class and sequence diagram.
  2. The student can: apply a number of specific characteristics of the C++ language such as:
     - Using polymorphism.
     - Implementing copy constructor.
     - Implementing assignment operator.
     - Creating a dynamic class and taking memory leaks into account.
     - The slicing problem.
     - Making a template class.
  3. The student can:
     - Finding errors in a program by using a debugger.
     - Realising and applying a unit test.
- Object-oriented programming part II.
  1. The student can: from a UML diagrams, implement C++ data structures and vice versa.
     - Implementing a class.
     - Implementing an association, aggregation and composition. Emphasising the multiplicity of 1 on multiples.
     - Applying algorithms from the STL library to the applied data structures.
     - Calling up various methods from a sequence diagram.
     - Applying exceptions in the C++ language.
     - From code to a class and sequence diagram.
  2. The student can: apply a number of complex specific characteristics of the C++ language such as:

- Applying algorithms to a data structure.
- Using functors.

3. The student can:
   - Finding errors in a program by using a debugger.

Because students had not yet worked structurally with the debugger at the start of the course ”Introduction OOP”, much attention was paid in this course to working with the debugger. During the lecture, many examples were given, and during the practical, the students had to demonstrate through screenshots that they had worked with the debugger.

The ”OOP part II” course focused more on advanced features such as exceptions and applying of STL data structures and algorithms, as a result of which the visualisation of objects became less important or even unclear. As an example, the fact that the STL datastructure ”set” works according to the principle of a red-black tree [31] is not directly relevant here. Besides, it is not easy to see this with a debugger. In order to show the advantages of the debugger as a learning tool, the lecture was converted into a seminar, in which the students had to make small assignments and explain them with the help of the debugger. Converting a lecture to a seminar was possible because the number of lectures was increased from one to two lecture hours per week.

3.1.2 The metrics used explained

This section briefly explains the metrics applied when comparing the exam to the level of demand.

- **Average** : This is mainly used to get an indication of whether the test has been done better than the previous test.
- **Variance**: This is used to get an indication of the spread of the results. If the spread varies too much, a histogram is used to get an impression of the distribution.
- **High score**: This is only used to determine whether there are students who have the question absolutely right.
- **M/Mmax**: This is one of the metrics that has been most commonly used during the analysis and has been used mainly to obtain an indication about how well a question or part of a question has been made. The M/Mmax is an indication of the difficulty of the question and is between 0 and 1, a 0 indicates that no one get the answer right, while a 1 indicates that everyone has the right answer to the question
- **Rasch UnIT (R_{it})**:This is also an important metric. It indicates as to whether the question is ambiguous. The R_{it} value means item-total correlation; this is the correlation between the scores of all students on question x and the total score of all students. If the R_{it} value of a question is between 0.35 and 1, it indicates that the students with a high score answer the question right and the students with a low score wrong. If the R_{it} value of a question is between -1 and -0.35, students with a low score answer the question right and students with a high score wrong [10].

- **Crombach's alpha**: This metric is only used to evaluate the exam as total and gives an indication of the reliability of the test. The value is between 0 (not reliable) and 1 (fully reliable) [10].
- **Mann-Whitney U test**: The Mann-Whitney U Test [40] was used to test the static significance between similar courses in two different years and is done with SPSS.

3.1.3 The participants of this research

The participants in this study are second-year students at The Hague University of Applied Sciences of the academy Information Technology & Design (ITD) study programme Network System Engineering (NSE). The students had followed courses C and microcontroller programming and the course object-oriented modelling. Although the students followed the first year and the first term of the second year, there is a fairly large difference in motivation and knowledge. It varies from students with an MBO without programming and mathematics, who do not like programming, to students who dropped out of the TU computer science.

Due to the diversity of the students and a group of about 50 students, it is not easy to create two equal groups and to teach one group the old education and the other the new education. Moreover, groups have already been formed in the first year; several students have known each other since the previous education. How to reach a reasonably reliable opinion whether the use of a debugger leads to a better result is described in the next section.

3.1.4 Measurements of the experiment

Testing a hypothesis can be done using a control experiment, in which one group participates in the experiment (the experiment group) and another group that does not (the control group) [4]. By comparing the data of both groups afterwards, it can be determined whether the hypothesis is correct or not. In order to be able to do a control experiment, an experiment and a control group are needed.

Unfortunately, in this experiment it was not possible to create two equivalent groups that independently of each other did the old course and the modified course in the same period. As described in Section 3.1.3, the diversity within the group of especially the pre-university courses is high compared to the number of students, besides only one lecture was planned for a group of students. Furthermore, the students would know from each other who was in which group, which would give an enormous amount of noise. That is why the renewed courses were offered to all students.

In order to test the hypothesis "*People who use a debugger when learning object-oriented programming better understand the concepts*", the results of both written exams (Introduction OOP, OOP Part II) were compared to the written exams of the previous year. To determine whether the current group of students has a similar level of knowledge as the groups of the past two years. The results of the previous courses "Object-oriented modelling"(OOMOD) and "microcontroller programming"(EMBED) have also been taken into the equation. In this way, an impression can be obtained as to whether the students are better or worse than the students of previous years.

The analyses of the exams were done at three-point. At first, the results of the exam were analysed itself using, among others, the cronbach alpha, Rasch UnIT (R_{it}), average and M/Mmax. The M/Mmax is the average score on a question (M) divided by the maximum score (Mmax) of the question. This index gives an indication of the easiness of the question [18]. At second, the results were analysed with the similar exam a year earlier. Finally, the results were analysed with the results of other courses in the same field, so that we got an indication of the group was better or worse than other groups. The analysis of the exam on itself and the comparisons with other exams are mainly done with the help of Excel and SPSS. More detailed information on the analysis of the results can be found in Chapter 4. Except for the analyse of the theoretical exam, the students also completed a questionnaire.

The questionnaire was designed to measure how students perceived the use of the debugger as an auxiliary tool for object-oriented programming in both lectures and lab-sessions. The questions about the lectures were divided in two parts, one to look back at the material of the previous lecture and one how the debugger was used during the explanation of new fabric. Both parts consisted of a few multiple-choice questions (mainly on a 5-points Likert scare) and two open questions, one about tips and one about flops. The questions about the lab-sessions were also multiple-choice and two open questions about tips and flop. The results of the questionnaire are displayed using boxplots.

Despite that it was not possible to divide the cohort into two groups, it was quite possible to get a reliable view of the results of the exam and therefore a good indication whether the hypothesis "People who use a debugger when learning object-orient programming better understand the concepts" should be accepted or rejected.

3.1.5 Choice of comparison exam

As described in Section 3.1.3, this experiment cannot involve a control group; it is mainly compared to a similar exam from a previous year. The experiment covers both the course "Introduction OOP" and "OOP part II". As mentioned before, most of the modifications take place in the course "Introduction OOP". For this reason, only the exams from previous years of the course "Introduction OOP" are considered.

In order to find out which particular parts of the course material the students had the most difficult to understand, several previous exams were analysed. First of all, an analysis was done of the exams of the past five years. Afterwards, the exam for last year was explicitly discussed. An extensive analysis was done on some questions, so with the help of the debugger extra attention could be paid to the badly performed parts.

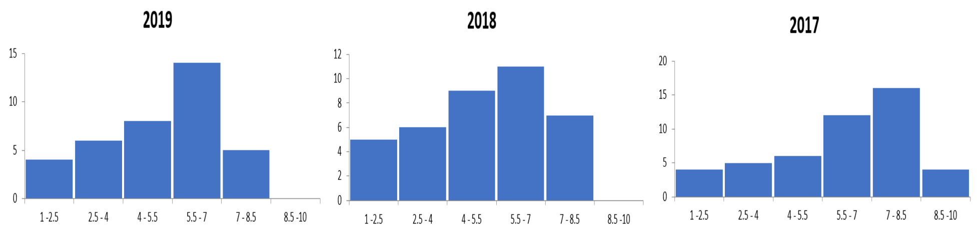
The course "Introduction object-oriented programming" has been taught in a dynamic environment. There used to be two departments Technical Information Technology, one in Delft and one in The Hague (main department). Both Technical Information Technology departments merged in 2014 and in 2017 merged five information technology departments to HBO-ICT. During the TI specialisation, the course had been taught in the fourth term of the foundation year, but during the HBO-ICT specialisation the course is being taught in the second year. Because of moving the course from the foundation year to the second year, there was no an exam in 2016. Because these changes have taken place, the values in Table 3.1 are not entirely comparable.

The final results of the exams are shown in Table 3.1. All of them are written exams at the similar level with open theoretical and skill question. To check if the exam had the similar level, a second lecture with knowledge about the content would check the exam. The exams consisted of different parts: in the first part there were some theoretical questions, in the second part there were some questions about implementations of methods. In the third part there was a question about the implementation of a template, and in the final part, there was often a question to implement a sequence diagram. The final score was based on ninety points for answering the questions add ten points as a bonus. At this way, the students pass for the exam with 50% of the score. The Cronbach's alpha [11, 10] of the

	Exam				
	HBO-ICT			TI	
	2019	2018	2017	2015	2014
Number of students	37	39	47	37	50
Success rate	51.4%	50%	70.2%	78%	60%
Average	5.2	5.1	6.1	6.5	5.5
Variance	3.6	3.7	4.9	3.2	4.3
Cronbach's alpha	0.8	0.8	0.85	0.85	0.84

Table 3.1: Globaal results of the exam introduction OOP.

exams vary between 0.8 and 0.85; this indicates that all tests were reliable. The success rate for the last two exams were about the same while the results of 2017 and 2015 were much higher. This might be explained by the fact that HBO-ICT started in 2017, and several students who had received a negative study advice from one of the forerunners were allowed to try again. The high result of 2015 might be explained by the fact that a large number of students had already dropped out (of the department TI location The Hague only 3 of the 20 students who participated in embedded programming half a year earlier were left). If we



(a) Exam 2019. (b) Exam 2018. (c) Exam 2017.

Figure 3.2: Distributing of the results of the exams Introduction object-orieted programming.

focus on the results of the HBO-ICT students, the distribution of final scores are plotted in a histogram such as shown in Figure 3.2. All three courses have approximately the same distribution. Furthermore, the 2017 exam appears seems to have a structural higher score, which is a confirmation of Table 3.1. Both the 2018 and 2019 exams are very similar in terms of the number of participants and the result. The difference between them falls under the noise. Because both Table 3.1 and Figure 3.2 show that there is little difference between the 2018 and 2019 examinations, the choice was made for the last examination taken as a control examination.

3.2 Analysis of the 2019 ”Introduction Object-Oriented programming” exam

The examination of the school year 2018-2019 has been analysed to get a good view of which elements of the subject of the lesson were not so well understood. Thirty-eight students took part in the exam, one of whom submitted an empty form. We cleaned up the data by removing the empty form. Finally, we included thirty-seven students in this analysis.

The exam consisted of twelve questions (A - L) based on a simple class diagram. Of the twelve questions, four were theory questions, seven about the implementation of a method and one question about the implementation of a sequence diagram. The maximum number of points for the twelve questions was ninety, with the ten-point bonus being a hundred points. The bonus of ten points is added so that the result is sufficient for a score of 50%. A second lecture with knowledge checked the exam. Table 3.2 shows the complete analysis and the analysis split per question of the exam. With a Crombach alpha, which is an indication of the reliability of the test, of 0.8, it is a reliable test [10].

	Final score	total	A	B	C	D	E	F	G	H	I	J	K	L
maximum number of points	10	90	5	5	5	5	5	5	10	10	10	10	10	10
total number of points		3330	185	185	185	185	185	185	370	370	370	370	370	370
total number of points by students		1555	40	86	112	77	158	136	175	162	59	247	157	146
avarage	5.2	42.0	1.1	2.3	3.0	2.1	4.3	3.7	4.7	4.4	1.6	6.7	4.2	3.9
variance	3.6	360.0	4.0	2.9	2.9	6.1	2.6	4.3	15.8	10.6	8.6	12.0	17.2	13.4
standard deviation	1.9	19.0	2.0	1.7	1.7	2.5	1.6	2.1	4.0	3.3	2.9	3.5	4.1	3.7
high score	8.2	72.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0	10.0	9.0	10.0	10.0	10.0
M/Mmax			0.2	0.5	0.6	0.4	0.9	0.7	0.5	0.4	0.2	0.7	0.4	0.4
r_{it}			0.2	0.4	0.4	0.4	0.6	0.6	0.7	0.8	0.5	0.7	0.6	0.6
Cronbach’s alpha		0.8												

Table 3.2: Analyse of the results of exam Jan 2019.

Other quality indicators that apply per question are the Means/max of the score (M/M-max) and the r_{it} value [10, 18]. The M/Mmax is an indication of the difficulty of the questions and is between 0 and 1, a 0 indicates that no one get the answer right, while a 1 indicates that everyone has the right answer to the question. The r_{it} value means item-total correlation, this is the correlation between the scores of all students on question x and the total score of all students. If the R_{it} value of a question is between 0.35 and 1, it indicates that the students with a high score answer the question right and the students with a low score wrong. If the R_{it} value of a question is between -1 and -0.35, students with a low score answer the question right and students with a high score wrong [10]. Furthermore, there are other indicators such as average, variance, standard deviation. First we look at the M/Mmax, then at the R_{it}. In the next section the questions A, B, G, H, I and J are analyzed.

In this exam, the questions A and I both have an M/Mmax value of 0.2, which indicates that these questions are difficult or unclear. Question A is a theoretical question about the keyword `virtual` and question I is a question about the copy constructor. Both questions are directly related to the semantics of object-oriented programming. The questions G and H have a R_{it} value of 0.7 and 0.8, the M/Mmax value are 0.5 and 0.4. This indicates that these questions are done underperformance, but they are distinctive. Question G is about using a vector and calling methods of objects. Question H is about implementing a derived constructor, and it is a clear example of object-oriented programming where insight into the understanding of the derived class is essential. Question J is an average question with a R_{it} value of 0.7 and an M/Mmax of 0.7. This question is about an overridden method and check if the method can be called, it is a (1 —— 0..1) relation.

Question E has an M/Mmax value of 0.9. This indicates that the question is too easy or it is a stating obvious question [10]. This question is about the implementation of the base class constructor, which indicates that the term ”constructor” is known to the students. All the questions of this exam have a positive R_{it} value, this indicates that there are no ambiguous questions [10]. Questions B to L all have a value greater than 0.35, which indicates they are distinctive. Only question A with a value of 0.2 is not sufficiently distinctive. This could be because it is a theoretical question where insight is less important.

With a Crombach alpha of 0.8 indicates that it was a reliable examination. Because most students did not sufficiently understand object-oriented programming, the average score is low. It is possible that the debugger can be used to understand the object-oriented concepts better.

3.2.1 Explanation the exam assignment

To obtain more information about which components the students did not understand, we thoroughly examined the last exam, so that more time can be spent on a problematic topic during lectures and the practical with the help of the debugger.

The exam was open book and built around a class diagram, which is shown in Figure 3.3. Concepts related to the base and derived class, such as the implementation of constructors and polymorphism, were tested with the base class **EmbeddedPlatform**, which has two inheritance classes. Working with a vector and the concept template was done with the class **Controller**, overloading of the copy constructor was done by the class **Raspberry** and calling methods of other classes was mainly done by methods of the classes **Controller** and **Raspberry**. The questions in which the debugger can have the most impact for a better understanding of OOP by students will be analysed in detail in the next subsections, further discussing what effect the debugger can have to clarify the concept of OOP. The complete exam is shown in Appendix B.

3.2.2 Question A and B.

Both questions are related to the pure virtual method *`virtual double readValue( int ) const =0;`*, question A to the keyword `virtual` and question B with pure virtual

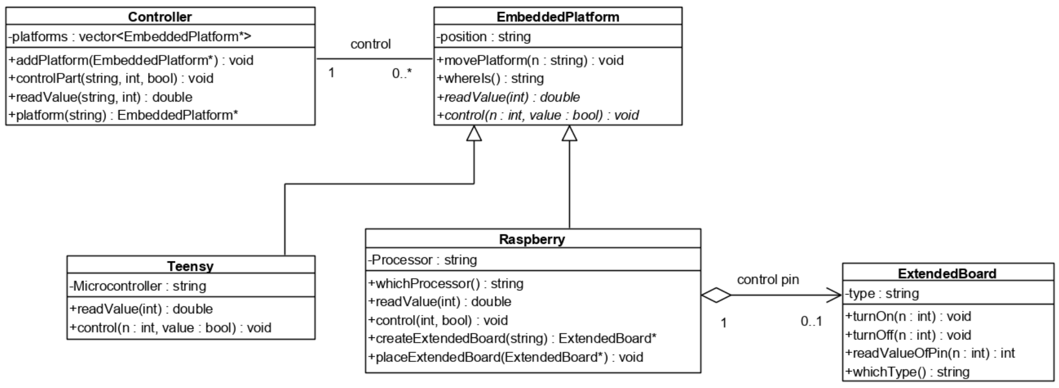

Controller
-platforms : vector<EmbeddedPlatform*>
+addPlatform(EmbeddedPlatform*) : void
+controlPart(string, int, bool) : void
+readValue(string, int) : double
+platform(string) : EmbeddedPlatform*
control
1
0..*
EmbeddedPlatform
-position : string
+movePlatform(n : string) : void
+whereIs() : string
+readValue(int) : double
+control(n : int, value : bool) : void
Teensy
-Microcontroller : string
+readValue(int) : double
+control(n : int, value : bool) : void
Raspberry
-Processor : string
+whichProcessor() : string
+readValue(int) : double
+control(int, bool) : void
+createExtendedBoard(string) : ExtendedBoard*
+placeExtendedBoard(ExtendedBoard*) : void
control pin
1
0..1
ExtendedBoard
-type : string
+turnOn(n : int) : void
+turnOff(n : int) : void
+readValueOfPin(n : int) : int
+whichType() : string

Figure 3.3: Class diagram of the exam January 2019.

`"=0"` of the class **EmbeddedPlatform**. Question A was made the worse of the exam and had an M/Mmax of 0.2 and a R_{it} of 0.2. The results are shown in Table 3.3.

	max points	average	variance	M/Mmax	R_{it}
general	5	1.1	4	0.22	0.2

Table 3.3: Analyse of questing A.

The R_{it} is low but still positive, the fact that the M/Mmax and the R_{it} are both low indicates that a lot of students do not understand the meaning of the keyword `virtual`, even the better students. As can be shown in Table 3.4, question B was done better than question A with an M/Mmax of 0.5 and an R_{it} of 0.4. Both questions could get better results by using the debugger during the lessons.

	max points	average	variance	M/Mmax	R_{it}
general	5	2.3	2.9	0.5	0.4

Table 3.4: Analyse of questing B.

To use the debugger to improve the results, the concept of pure virtual function will first be explained. A pure virtual method means that the function is abstract, and therefore, the class is abstract, and no object of that abstract class can be created [51]. To better understand the concept of `virtual`, the ”step into” function has been used of the debugger’s tracer. When in Code fragment 3.1 a breakpoint is set to line 5, the debugger will jump to the `readvalue` method of the class **Raspberry**. If the debugger is set to line 6, the debugger will jump to the `readvalue` method of the class **Teensy**.

```
1  Raspberry pi("ARM␣Cortex␣A7","living␣room");
2  Teensy ts("ARM␣CortexM4","hall");
3  EmbeddedPlatform * embed1=&pi;
4  EmbeddedPlatform * embed2=&ts;
5  embed1->readValue(2);
6  embed2->readValue(2);
```

Code fragment 3.1: Explaination for the *virtual* keyword.

By applying the debugger in this way during lectures, the score might be better on this question. Further explanation on applying the debugger to overloading and overriding can be found in Section 2.2.3.

3.2.3 Question G.

With this question the method `double readValue (string s, int p) const;` of the class **Controller** had to be implemented. The parameters are the position of the embedded platform and the physical pin that should be read. With this question the following points are tested.

1. Using a vector.
2. Calling methods of objects.

This question is divided into three OOP parts and one remainder. The remainder refers to minor errors that have little to do with concepts of object-oriented programming, such as ";" forgotten, writing errors, etcetera. The results of these parts are shown in Table 3.5. The most interesting parts about OOP of this question are both method invocations, and

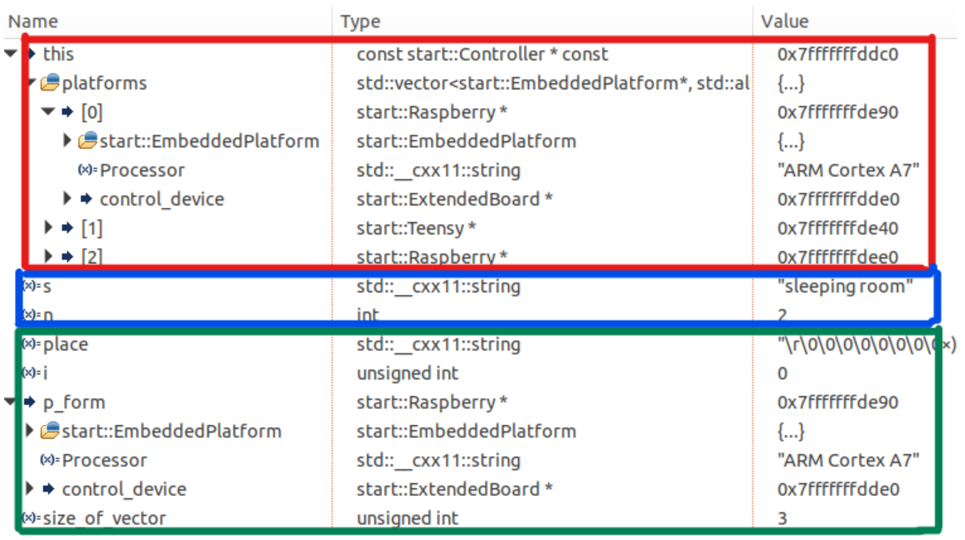
	max points	average	variance	M/Mmax	R_{it} question	R_{it} exam
general	10	4.73	15.81	0.47		0.67
looping the vector	2	0.95	0.89	0.47	0.85	
calling method `whereIs`	3	1.24	1.91	0.41	0.94	
calling method `readValue`	3	1.78	1.56	0.59	0.87	
diverse/remainder	2	0.76	0.80	0.38	0.89	

Table 3.5: Analyse of questing G.

especially the method `whereIs`. This method of the class **EmbeddedPlatform** has to be called to get the value of the attribute position. This method has the lowest M/Mmax value but it is the most distinctive part of this question with a R_{it} value of 0.94. Because the R_{it} value was high on each part (close to 1), it only indicated that probably the better students could answer the question [10]. The challenge was whether the debugger would help the students to understand the issue better.

To clarify looping through the vector and invoking the method by using the debugger, the tracer and the variable screen have to be used.

The code of fragment 3.2 is one of the possible answers to this question. Clarifying this code with the help of a debugger could be done by using the tracer and watching the variables as can be seen in Figure 3.4. By showing the variables, it becomes clear that the attribute `platforms`, which is a vector, has three-pointers (addresses) to different embedded-platforms. It also becomes clear that a vector looks like an array.



Name	Type	Value
this	const start::Controller * const	0x7fffffffddc0
platforms	std::vector<start::EmbeddedPlatform*, std::al	{...}
[0]	start::Raspberry *	0x7fffffffde90
start::EmbeddedPlatform	start::EmbeddedPlatform	{...}
Processor	std::__cxx11::string	"ARM Cortex A7"
control_device	start::ExtendedBoard *	0x7fffffffdde0
[1]	start::Teensy *	0x7fffffffde40
[2]	start::Raspberry *	0x7fffffffdee0
s	std::__cxx11::string	"sleeping room"
n	int	2
place	std::__cxx11::string	"\r\0\0\0\0\0\0\0\0\(x)
i	unsigned int	0
p_form	start::Raspberry *	0x7fffffffde90
start::EmbeddedPlatform	start::EmbeddedPlatform	{...}
Processor	std::__cxx11::string	"ARM Cortex A7"
control_device	start::ExtendedBoard *	0x7fffffffdde0
size_of_vector	unsigned int	3

Figure 3.4: Value of the variables, with breakpoint at line 8.

```
1  double start::Controller::readValue(string s,int n) const {
2     EmbeddedPlatform *p_form;
3
4     unsigned int size_of_vector = platforms.size();
5
6     for(unsigned int i=0;i < size_of_vector;i++) {
7          p_form = platforms[i];
8          string place = p_form->whereIs();
9          if( place == s)
10              return platforms[i]->readValue(n);
11     }
12     return NOPLATFORM;
13 }
```

Code fragment 3.2: An implementation of the method *readvalue*.

If the tracer is on line 8, the pointer `p_form` refers to the first object in the array and calls the method *`whereIs`* of that object. In this way, the debugger could be used to clarify specific OOP characteristics such as calling methods and using a vector.

3.2.4 Question H.

With this question the constructor of the derived class **Raspberry** had to be implemented. The points on which we assessed are:

- Calling constructor of the base class **EmbeddedPlatform** (4 points).
- Initialize attribute `Processor` (2 points).
- Initialize pointer `control_device` to NULL (3 points).
- The remainder (1 point).

The students did not perform well on this question when observing the M/Mmax score 0.44, but it had the highest R_{it} value of the exam, which indicates that ”good” students score well, and the ”bad” students score worse at this question. When we look at the individual items, which are shown in Table 3.6, we notice that the initialization of the attribute processor with an M/Mmax of 0.84 was done well by most students, and that the initialization of the base class and the association to the extended board (pointer to `control_device`) were done badly. The initialization of the base class has a R_{it} of 0.86, which indicates that only good students perform well on this part of the question. For our study, the question thus was, how can the debugger be used so that not only the best students could create this question?

	max points	average	variance	M/Mmax	R_{it} question	R_{it} exam
general	10	4.38	10.58	0.44		0.74
Calling base constructor	4	1.57	3.64	0.39	0.86	
Initialize attribute `Processor`	2	1.68	0.61	0.84	0.63	
Initialize pointer `control_device`	3	0.76	1.58	0.25	0.57	
diverse/remainder	1	0.38	0.24	0.38	0.79	

Table 3.6: Analyse of questing H.

With question H, two important principals of object-oriented programming were tested.

1. The base class is a part of the derived class, so if the derived class is created, the base class has to be created too.

2. All the attributes have to be initialised.

The constructor of the class **Raspberry** was called in the main code with the statement as is shown in Code fragment 3.3.

```
Raspberry pi("ARM␣Cortex␣A7","living␣room");
```

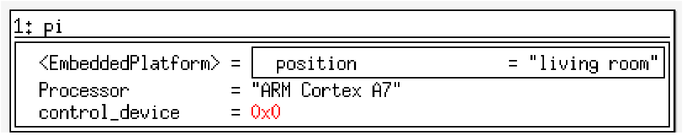
Code fragment 3.3: Calling the constructor of the class Raspberry.

With the tracer of the debugger and the ”step in” possibility, it can be shown that the base class is a part of the derived class. At first, the debugger jumps to the constructor of the derived class (line 2 of the Code fragment 3.4), then the debugger jumps to the constructor of the base class (line 6) where the attribute position is initialized. Then the debugger turns back to the constructor of the derived class and initializes the attributes `processor` and `control_device`.

```
1
2  start::Raspberry::Raspberry(string m,string p):EmbeddedPlatform(p),Processor(m),
3                                                                  control_device(0) {
4  }
5
6  start::EmbeddedPlatform::EmbeddedPlatform(string p):position(p) {
7  }
```

Code fragment 3.4: The constructor of the class Raspberry.

By using the DDD debugger, the underlying relationships between the base class and the derived class can be visualized, which is shown in Figure 3.5. The name of the base class is shown between <> with the attributes enclosed in a rectangle behind it.



```
1: pi
<EmbeddedPlatform> = | position              = "living room"|
Processor          = "ARM Cortex A7"
control_device     = 0x0
```

Figure 3.5: Visualization of the object pi of class Raspberry.

The debugger can be helpful to clarify the relationships between the base class and the derivative class. More information about applying the debugger and the constructor of a derived class can be found in Section 2.2.3

3.2.5 Question I.

With this question the copy constructor of the derived class **Raspberry** had to be implemented. The points on which we assessed are:

- Write down an explanation why a copy constructor is needed (3 points).
- Calling constructor EmbeddedPlatform (2 points).
- Initialize attribute (1 points).
- If there is a pointer to extendedBoard, use new ExtendedBoard (using statement new) (3 points).
- The remainder (1 point).

The students performed worst on this question of the exam with an M/Mmax of 0.16. The variance of 8.64 and a R_{it} of 0.55 indicates that most students score very low on this question, but the question is distinctive. When we look at the individual items, which are shown in Table 3.7, we see that every item scores low. The question that arose here: Can the principle of the copy constructor be clarified by using the debugger tool so that not only the better students understand the concept of the copy constructor?

	max points	average	variance	M/Mmax	R_{it} question	R_{it} exam
General	10	1.59	8.64	0.16		0.55
Explanation	3	0.68	1.5	0.23	0.93	
Calling base constructor	2	0.32	0.56	0.16	0.74	
Initialize attribute	1	0.16	0.14	0.16	0.74	
make new platform	3	0.38	0.74	0.13	0.94	
diverse/remainder	1	0.05	0.05	0.05	0.53	

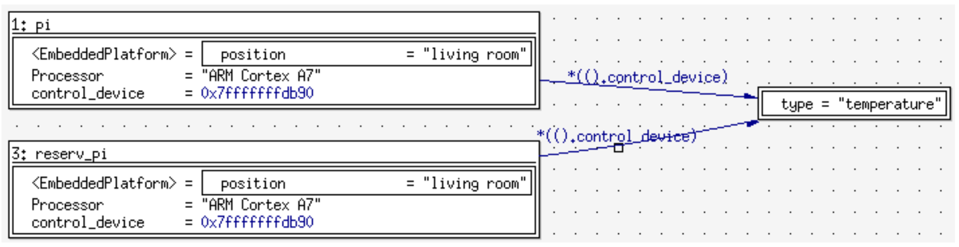
Table 3.7: Analyse of questing I.

To answer this question, the students have to know what a copy constructor is and when they have to implement it. The essence of this question was, what goes wrong with the statement ”`Raspberry reserv_pi(pi);`”.

```
1  ExtendedBoard extb("temperature");
2  Raspberry pi("ARM␣Cortex␣A7","living␣room");
3  pi.placeExtendedBoard(&extb);
4  Raspberry reserv_pi(pi);
```

Code fragment 3.5: Copy the object pi.

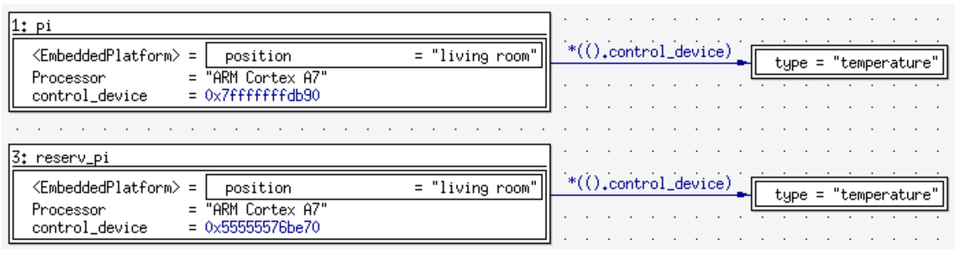
Code fragment 3.5 shows the creation of the objects of the classes ExtendedBoard and Raspberry (line 1 and 2), line 3 the association between the raspberry and the extended board and line 4 the copy of the object `pi`. When an object is copied, all attributes are also copied,



```
1: pi
  <EmbeddedPlatform> = | position              = "living room" |
  Processor          = "ARM Cortex A7"
  control_device     = 0x7fffffffdb90

3: reserv_pi
  <EmbeddedPlatform> = | position              = "living room" |
  Processor          = "ARM Cortex A7"
  control_device     = 0x7fffffffdb90

*(().control_device)    type = "temperature"
*(().control_device)
```

Figure 3.6: Copy of Raspberry without copy constructor.

so the copied object gets a reference to the same extended board as the original object. This is represented by the DDD debugger, as shown in Figure 3.6. If it is desirable that every copied raspberry also gets its extended board, the copy constructor must be overloaded.



```
1: pi
  <EmbeddedPlatform> = | position              = "living room" |
  Processor          = "ARM Cortex A7"
  control_device     = 0x7fffffffdb90
                                        *(().control_device)    type = "temperature"

3: reserv_pi
  <EmbeddedPlatform> = | position              = "living room" |
  Processor          = "ARM Cortex A7"
  control_device     = 0x55555576be70
                                        *(().control_device)    type = "temperature"
```

Figure 3.7: Copy of Raspberry with copy constructor.

The result of overloading the copy constructor is shown in Figure 3.7. To show that the copy constructor is called, a breakpoint can be set at line 4, and with the tracer and the ”step in” possibility, it is shown that the copy constructor is actually called. Because with the DDD debugger the relationships between the objects can be visualized in combination with the tracer, the debugger could help to clarify the copy constructor. More information about applying the debugger and the copy constructor can be found in Section 2.2.2

3.2.6 Question J.

The method `void control (int n, bool v);` of the class **Raspberry** should be implemented. This question aims to check the 0..1 association and to call methods of the association class. The points on which we assessed are:

- Control if device is present. (3 points).

- Control if the pin has to be on or off (2 points).
- Calling TurnOn method of class **ExtendedBoard** (3 points).
- Calling TurnOff method of class **ExtendedBoard** (3 points).
- The remainder (1 point).

This question is done relatively well, with an M/Mmax of 0.67; this is above the average of the exam (M/Mmax=0.47). With a R_{it} of 0.73, it indicates that it is a distinctive question. When we look at the individual items, which are shown in Table 3.8, checking whether an extended board is present is the least done. The R_{it} of calling the methods of the extend-

	max points	average	variance	M/Mmax	R_{it} question	R_{it} exam
	10	6.68	12.00	0.67		0.73
Control device	2	1.11	0.93	0.55	0.77	
Control pin on or off	1	0.86	0.12	0.86	0.70	
Turn pin on	3	2.00	1.22	0.67	0.95	
Turn pin off	3	1.95	1.33	0.65	0.96	
diverse/remainder	1	0.76	0.19	0.76	0.72	

Table 3.8: Analyse of questing J.

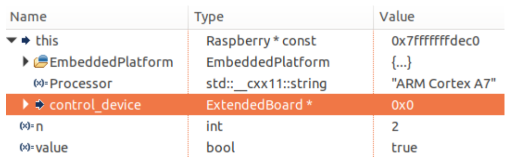
edBoard class is high. This indicates that almost every student who did this question well, could also call the methods of the extendedBoard class. Although this question has been answered relatively well, a higher M/Mmax would be desirable. We hypothesize that students will perform better on this question if the debugger is used during class. In particular, consider Code fragment 3.6, in which we call the method `control` before and after the extended board is placed.

```
1  ExtendedBoard opz("temperature");           //extended board type temperature
2  Raspberry pi("ARM␣Cortex␣A7","living␣room");//raspberry with processor ARM Cortex A7,
3                                              // will be placed in living room
4  pi.control(2,true);
5  pi.placeExtendedBoard(&opz);
6  pi.control(2,true);
```

Code fragment 3.6: Calling the method `control` with and without extended board.

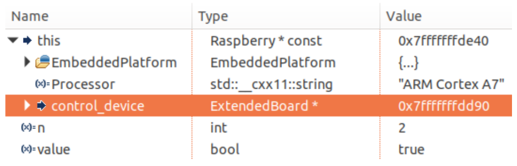
By placing breakpoints where the method `control` is called (line 4 and 6), and using the "step in" function to jump into the method `control`, the attributes of object `pi` can be watched. By watching the variables, we can show the difference between with or without an extended board as is shown in Figure 3.8. Furthermore, we can show why a test to check whether an extended board is present on this needed or not, namely the occurrence of a NULL pointer exception. Clarifying to call a method of a different class could be done by using the tracer and the "step into" possibility.

After the tracer has jumped into the method `control`, it is checked whether an extended board is present and on the value of the parameter `value`. With the next step (Code



Name	Type	Value
this	Raspberry * const	0x7fffffffdec0
EmbeddedPlatform	EmbeddedPlatform	{...}
Processor	std::__cxx11::string	"ARM Cortex A7"
control_device	ExtendedBoard *	0x0
n	int	2
value	bool	true

(a) Without extended board.



Name	Type	Value
this	Raspberry * const	0x7fffffffde40
EmbeddedPlatform	EmbeddedPlatform	{...}
Processor	std::__cxx11::string	"ARM Cortex A7"
control_device	ExtendedBoard *	0x7fffffffdd90
n	int	2
value	bool	true

(b) With extended board.

Figure 3.8: Check whether an extended board is available.

fragment 3.7, line 4 or 6), the tracer jumps to the method `turnOn` or `TurnOff` of the class **ExtendedBoard**.

```
1  void Raspberry::control(int n, bool value) {
2      if(control_device) {
3          if(value)
4                control_device->turnOn(n);
5          else
6                control_device->turnOff(n);
7          }
8  }
```

Code fragment 3.7: The method `control`.

The debugger can be used to clarify why it is necessary to test whether or not a method can be called from an external class.

3.2.7 Conclusion

The students had most problems with questions that needed insight, such as question H (constructor of derived class), question I (the copy constructor), question A (the keyword virtual) and also with communication between de classes(calling a method). We hypothesize that these concepts can be clarified by using a debugger. Many students knew the basic concepts as the constructor.

3.3 Analysis of the 2019 ”Object oriented programming part II” exam

The exam Object-oriented programming part II was just like the exam Introduction object-oriented programming based on a defined class diagram. In contrast to exam Introduction object-oriented programming, this exam did not have theoretical questions and was built around a more complex class diagram, which shows in Figure 3.9. The exam had two questions about implementation of classes and six questions about implementation of methods. The main topics were calling methods, exceptions and using the STL. The complete exam can be found in appendix D.

The results of the exam are shown in Table 3.9. With a Cronbach Alpha of 0.7, the exam can be regarded as satisfactory [11]. This indicates that the results of the exam are valuable. The total score of the questions is 80, and the final score is 10 + 9/8 * (overall score). The 10 points were needed to pass the exam with 50% good answers. The reason

	total	1	2	3	4	5	6	7	8
maximum number of points	80	12	5	20	6	6	17	6	8
total number of points	3520	528	220	880	264	264	748	264	352
total number of points by students	1965	347	164	461	182	111	390	149	161
avarage	44.7	7.9	3.7	10.5	4.1	2.5	8.9	3.4	3.7
variance	281.4	7.9	1.9	33.6	1.8	2.4	36.9	4.5	8.5
standard deviation	16.8	2.8	1.4	5.8	1.3	1.6	6.1	2.1	2.9
high score	69.0	12.0	5.0	18.0	6.0	5.0	17.0	6.0	8.0
M/Mmax		0.7	0.7	0.5	0.7	0.4	0.5	0.6	0.5
r_{it}		0.5	0.6	0.7	0.2	0.7	0.9	0.8	0.6
Cronbach’s alpha	0.7								

Table 3.9: Analyse of the results of exam part2 April 2019.

that the overall rating is not 90 had to do with the distribution of the mutual points. The most interesting questions to analyse are those with a low M/Mmax and a high R_{it}. This indicates that few students were able to do these questions well and they did the exam well. So the question was probably unambiguous, but many students did not do well this question. We hypothesize that students will perform better on these subjects if the debugger is used during the classroom. In particular, the questions three, five and six will be discussed in the next part so that we can see which items could be clarified with the help of the debugger.

3.3.1 Explanation of the exam assignment.

The questions all relate to the defined class diagram, which is shown in Figure 3.9. The items to be tested are: exceptions, a base and derived class, an interface, calling methods and applying STL data structures. In addition to this class diagram, a main is also given on the exam, in which, for instance, various objects are created; the complete exam can be found in appendix D. In this way, the students are tested from class diagram to C++ code.

3.3.2 Question 3.

Question three is the most extensive question, in which twenty out of eighty points can be earned. The question is:
The method ’`bool turnOnOff();`’ of the class **Switch**, calls the method ’`void signal(Switch*,bool);`’ of the class **Controller**. The method ’`signal`’ can generate a **SwitchNotConnected** exception if the switch is not known to the controller. If the exception occurs, the room where it is located will be shown on display. The main items of this question were:

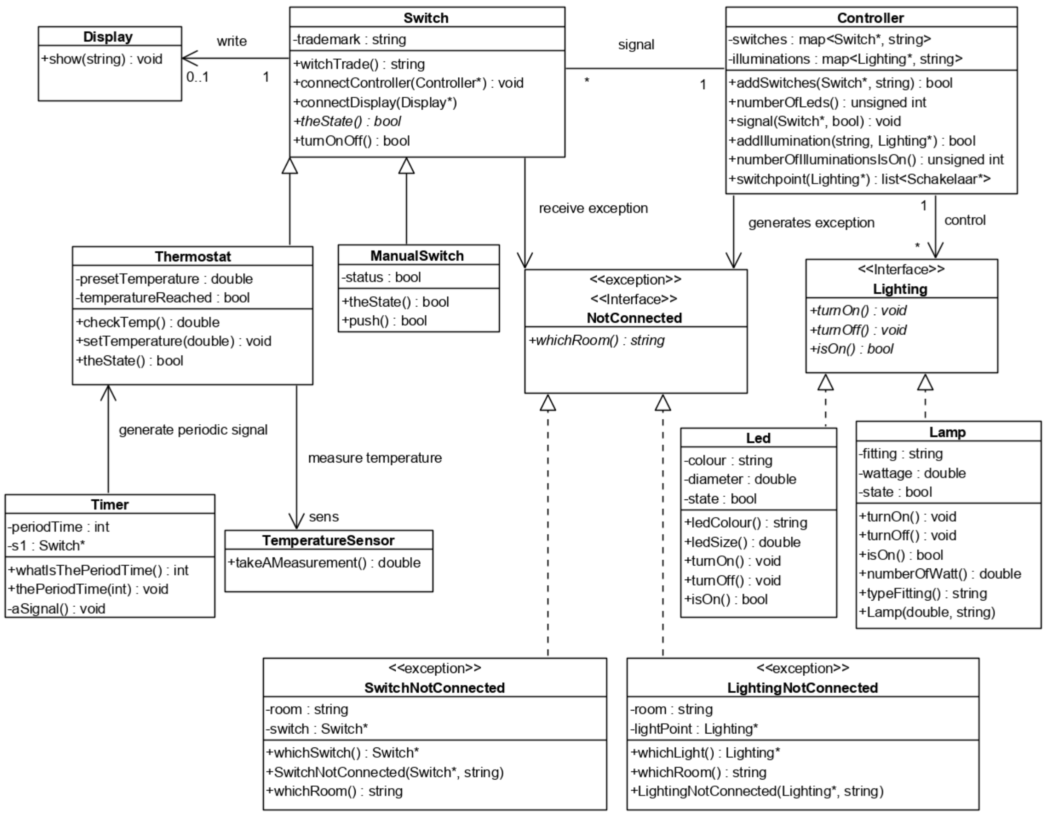


Figure 3.9: Class diagram of the exam March 2019.

- Calling the method `signal` with the `this` pointer and the result of the abstract function `bool theState()` as parameter.
- Catching of a possible exception.
- Calling the `show` function of the class **Display**.

Each item is subdivided into sub-items with each sub-item assigned several points. The distribution of points for this question is shown in Table 3.10. What is noticeable, that calling the method `show` of the class **Display** (line 8 and 9 of Code fragment 3.8) does not score well and in the particular case the check (line 8 of Code fragment 3.8), if a display is connected to the switch. The invocation of the method signal was proper, but the parameters (this pointer and the return value of the abstract method theState) were less good. The exception was proper, but it could be better. The result of this question was rather weak with an M/Mmax of 0.5, with typical C++ statement such as exception appearing as the most effective.

For our study, the question was: could we use the debugger during the lectures so that the subjects of this question would be better done. Code fragment 3.8 shows a probable answer to question three of this exam. If line 8 is commented on and a breakpoint is set on

		max points	average	variance	M/ Mmax	R_{it} question	R_{it} exam
		20	10.5	33.6	0.5		0.7
Method *signal*	Method invocation	3	2.3	1.4	0.8	0.8	
	this pointer	3	1.8	2.0	0.6	0.8	
	call function theState	3	1.7	1.6	0.6	0.8	
exception	try	2	1.4	0.9	0.7	0.8	
	catch	3	2.1	1.7	0.7	0.8	
method show of Display	check is available	2	0.1	0.2	0.1	0.0	
	call method show	2	0.9	0.7	0.5	0.7	
diverse/remainder		2	0.3	0.4	0.2	0.4	

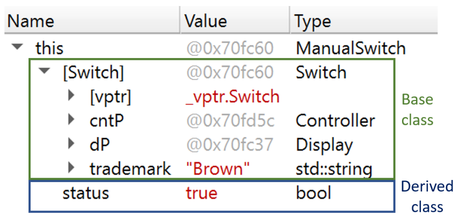
Table 3.10: Analyse of question 3.

line 9, the tracer can show very nicely what happens if `dp` has the value 0. An additional advantage is that we can demonstrate once more that an attribute always has to be initialized by the constructor.

```
1   bool  Switch::turnOnOff() {
2         if (!cnt)  //pointer to controller
3                return false;
4         try {
5               cnt->signal(this,theState());
6
7         }catch(NotConnected& nc) {
8               if(dp)  //pointer to display
9                  dp->show( nc.whichRoom());
10              return false;
11        }
12        return true;
13  }
```

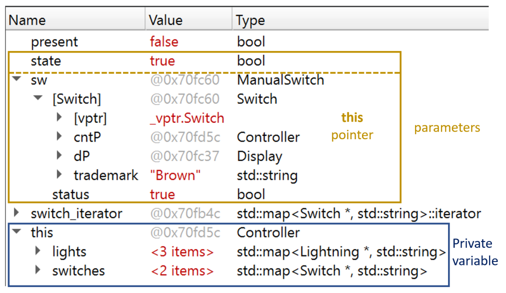
Code fragment 3.8: One answere to question 3.

Clarifying the `this` parameter can be done by the tracer in combination with watching the attribute values. Set a breakpoint on line 5 and show the attributes of the current object as shown in Figure 3.10a . Then we make a jump with the tracer into the `signal` function and show the contents of the parameter type `switch*`, this is the same as the previous object as shown in Figure 3.10b. Clarifying the try and catch can be done by placing a random



Name	Value	Type
this	@0x70fc60	ManualSwitch
[Switch]	@0x70fc60	Switch
[vptr]	_vptr.Switch	
cntP	@0x70fd5c	Controller
dP	@0x70fc37	Display
trademark	"Brown"	std::string
status	true	bool

(a) Values of the attributes before jump into the method `signal`.



Name	Value	Type
present	false	bool
state	true	bool
sw	@0x70fc60	ManualSwitch
[Switch]	@0x70fc60	Switch
[vptr]	_vptr.Switch	
cntP	@0x70fd5c	Controller
dP	@0x70fc37	Display
trademark	"Brown"	std::string
status	true	bool
switch_iterator	@0x70fb4c	std::map<Switch *, std::string>::iterator
this	@0x70fd5c	Controller
lights	<3 items>	std::map<Lightning *, std::string>
switches	<2 items>	std::map<Switch *, std::string>

(b) Values of the variables inside the method `signal`.

Figure 3.10: Showing the `this` parameter.

statement such as a `cout` on line 6 and breakpoints on lines 5 and 6. By doing a 'Step Over' and the `signal` function does not generate an exception, it jumps to line 6. If the signal function generates an exception, it will jump to line 7 and do the `catch` with `nc` as the object that has been thrown. By using an example such as Code fragment 3.8 during lectures, the debugger could help to understand the content of this question better.

3.3.3 Question 5.

With this question, the method *`bool addSwitch(Switch* s, string place);`* of the class **Controller** had to be implemented. The parameters were a pointer to the switch and the position where it had to be placed. The main items of this question were:

- Checking if the switch *`s`* is already in the map *`map<Switch*,string> switches;`*.
- Insertion the switch *`s`* in the *`map<Switch*,string> switches;`*.
- Diverse/remainder.

Often the operator '[ ]' is used to insert the key and associated data into the map. The disadvantage of using the operator '[ ]' is that it will overwrite the data if the key is already in the map. So by using the operator '[ ]' to insert a key, it needs to check, whether the data is already present. This is shown in Code fragment 3.9.

```
map<Switch*,string>::iterator it=switches.find(s);
if(it!= switches.end()) return false;

switches[s]=p;
return true;
```

Code fragment 3.9: Insertion in a map by using operator [].

A better way to insert into a map is by using the insert method of the class-map and check the return value, as is shown in Code fragment 3.10.

```
pair< map<Switch*,string>::iterator,bool>it;
it=switches.insert(map<Switch*,string>::value_type(s,p));
return it.second;
```

Code fragment 3.10: Insertion in a map by using the insert method.

Although we have accepted both possibilities, the question was not well done with an M/Mmax of 0.4. But the question was distinctiveness with a R_{it} of 0.7. Table 3.11 shows the results of question per partial. It shows that the map insertion check is wrongly done with an M/Mmax of 0.2.

We hypothesize that the usefulness of checking whether a switch is already present in the map can be clarified by using a debugger. A possibility is by watching both the keys and the data of the map before and after insertion. Set a breakpoint before insertion in the map and watch the content of the map, then do a 'step over' The content of the map will be changed: by changing a consist variable if the key is already in the map or by insertion the key s with the variable p.

	max points	average	variance	M/Mmax	R_{it} question	R_{it} exam
	6	2.5	2.4	0.4		0.7
Checking if the switch *s* is already in the map	2	0.3	0.6	0.2	0.6	
Insert into the map	3	1.8	0.9	0.6	0.8	
Return value + remainder	1	0.4	0.3	0.4	0.6	

Table 3.11: Analyse of question 5.

3.3.4 Question 6.

With this question, the method *`void signal(Switch* s, bool b);`* of the class **Controller** had to be implemented. The parameters were a pointer to a switch and the value that determined whether the light should be turned on or off. If the switch was not present, an exception `SwitchNotConnected` would be thrown. If the switch was present, but in the same room was not a light, an exception `SwitchNotConnected` would be thrown. The main items of this question were:

- To search the switch `s` is in the map `map<Switch*, string >switches;`.
- Throw exception.
- Use STL iterators.
- Diverse/remainder.

The question was not done great with an M/Mmax of 0.5, but it was very distinctive with a R_{it} of 0.9, which was confirmed with a high variance. The partial results of Table 3.12 shows that the students found it hard to use the STL iterators. If we want to clarify the subjects:

	max point	average	variance	M/ Mmax	R_{it} question	R_{it} exam
	17	8.9	36.9	0.5		0.9
searching the room in the ”switch” map	3	1.9	1.9	0.6	0.7	
throw excepion	5	3.5	4.3	0.7	0.9	
use STL iterators	7	3.2	9.5	0.5	0.9	
diverse/remainder	2	0.3	0.5	0.2	0.6	

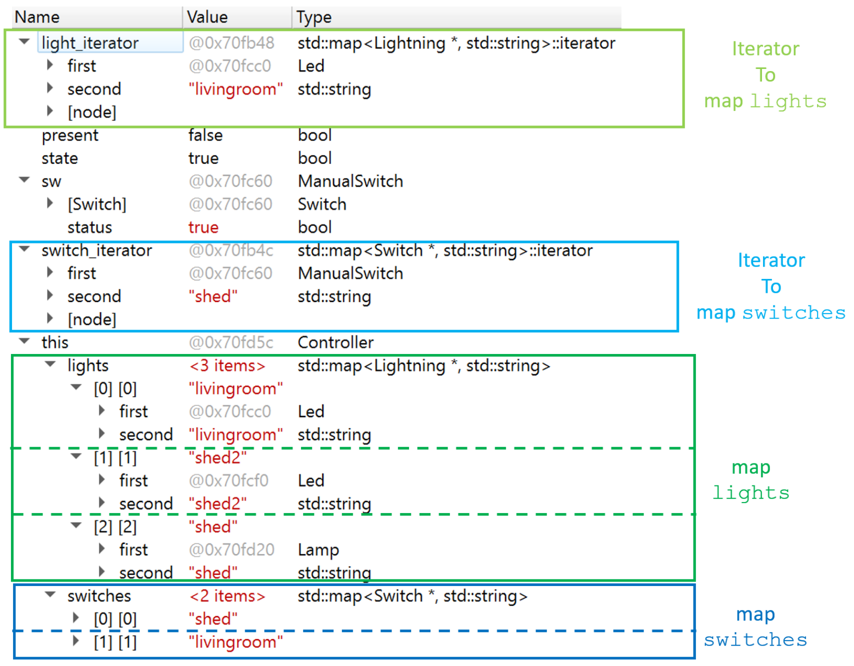
Table 3.12: Analyse of question 6.

applying an iterator and throwing an exception, we could use the debugger’s tracer. When we look at Code fragment 3.11, we see two iterators, both referring to a folder. By setting a breakpoint on line 11, the content of the map `lights` can be viewed, and the content to which the iterator `light_iterator` refers can be viewed too.

```
1  void Controller::signal(Switch* s,bool b) {
2     bool available=false;
3     map<Switch*,string>::iterator switch_iterator;
4     switch_iterator=switches.find(s);
5     if(switch_iterator == switches.end())
6         throw SwitchNotConnected(s,"unknown");
7
8     for(map<Lightning*,string>::iterator light_iterator(lights.begin());
9                             light_iterator != lights.end();light_iterator++) {
10       if(light_iterator->second == switch_iterator->second) {
11            available=true;
12            if(b)
13               light_iterator->first->zetAan();
14            else
15               light_iterator->first->zetUit();
16       }
17    }
18    if(!available)
19         throw SwitchNotConnected(s,switch_iterator->second);
20 }
```

Code fragment 3.11: Implementation of the method `signal`.

The same goes for the map `switches` and the iterator `switch_iterator`. Viewing the variables is shown in Figure 3.11. By walking through the method step by step with the tracer, it will become visible that the iterator refers to the next element in the folder.



```
Name                 Value          Type
▾ light_iterator     @0x70fb48      std::map<Lightning *, std::string>::iterator
    ▸ first          @0x70fcc0      Led
    ▸ second         "livingroom"   std::string
    ▸ [node]
  present            false          bool
  state              true           bool
▾ sw                 @0x70fc60      ManualSwitch
    ▸ [Switch]       @0x70fc60      Switch
      status         true           bool
▾ switch_iterator    @0x70fb4c      std::map<Switch *, std::string>::iterator
    ▸ first          @0x70fc60      ManualSwitch
    ▸ second         "shed"         std::string
    ▸ [node]
▾ this               @0x70fd5c      Controller
    ▾ lights         <3 items>      std::map<Lightning *, std::string>
        ▾ [0] [0]    "livingroom"
            ▸ first  @0x70fcc0      Led
            ▸ second "livingroom"   std::string
        ▾ [1] [1]    "shed2"
            ▸ first  @0x70fcf0      Led
            ▸ second "shed2"        std::string
        ▾ [2] [2]    "shed"
            ▸ first  @0x70fd20      Lamp
            ▸ second "shed"         std::string
    ▾ switches       <2 items>      std::map<Switch *, std::string>
        ▸ [0] [0]    "shed"
        ▸ [1] [1]    "livingroom"
```

Figure 3.11: Watching content of maps and iterators which refers to the map.

One of the features of throwing an exception is that the method is also immediately abandoned. This phenomenon can be demonstrated by setting a breakpoint on line 8. If an exception is thrown on line 6, the breakpoint will not be reached.

3.3.5 Conclusion

The students still had problems with calling the methods, with using STL features such as iterators and the different way of implementing such as inserting in a map. However, there were not many problems with exceptions. In general, there were fewer problems with this exam than with the exam of the course Introduction object-oriented programming.

3.4 Adjusting the education material

This section describes which adjustments have been done to the course material of both OOP courses. The adjustments to the teaching material are based on the need to involve the debugger more in the teaching so that it could be determined whether the hypothesis "*People who use a debugger when learning object-oriented programming better understand the concepts*" should be accepted or rejected. The analysis of both object-oriented programming exams have been taken into account in the adjustments of the courses. At first, the structure of both courses is discussed. Secondly, the adjustments made in the course Introduction object-oriented programming and finally, the adjustments made in the course Object-oriented programming part II.

3.4.1 Adjustments to the course structure

Both OOP courses consisted of a theoretical and a practical part which were taught in seven weeks, week eight was overrun, and the exam was in week nine.

The course Introduction OOP. This course consisted of two hours of lecture (90 minutes) and two hours of practical lessons. The teaching material of both components consisted mainly of implementing UML statements such as class and sequence diagram into C++ code. Furthermore, specific C++ topics such as operator overloading and a simple form of memory management were covered. A lecture consisted of a PowerPoint presentation in which interaction with the classroom as is desired. But no attention was paid to the debugger during the lecture. The practical part consisted of four assignments. The first assignment was with the tracer walking step by step through a given programme and to turn a LED on and off. The second and third were larger assignments in which classes and objects had to be created based on a UML diagram. The fourth assignment was an introduction to unit testing. The hope was that the students would use the debugger during the second and third assignment, which was practically not done.

To integrate the debugger tool in the new course "Introduction OOP", both the lecture and the lab sessions have been adapted. To keep the lecture attractive, we have decided to revise the previous college in the form of multiple-choice questions, in which the debugger plays a prominent role. To ensure that the students also work with the debugger during the larger assignments of the lab, we decided that screenshots of the debugger had to be submitted. More information about adapting the lecture and the practicals can be found in Section 3.4.2.

The course OOP part II. This course consisted of one hour of lecture (45 minutes) and an average three hours of practical lessons per week. The teaching material of both parts consisted mainly of converting class and sequence diagram into C++ code, with exception and the STL data structure as the main topics. A lecture consisted of a PowerPoint presentation in which interaction with the classroom is desired. But no attention was paid to the debugger during the lecture. The practical part consisted of five assignments where the students could use the debugger to detect errors. In practice, however, the "printf" method was used a lot.

Because the lessons of the new course were standardised to two hours of lecture and 2 lecture hours (90 minutes) of practical, we decided to use the extra hour of lecture to let the students make small assignments and ask one or two students to explain their solutions to the other students with the help of the debugger. More information about adapting the lecture can be found in Section 3.4.3. The practicum did not change, the students received less guidance.

3.4.2 Adjustments the course Introduction object-oriented programming

In order to introduce the debugger during lectures, capacity had to be created in the lectures. To make the object-oriented concepts clearer, the debugger would apply to both the review of the previous lecture and the current lecture. From the lecture schedule (Appendix G), it appeared that the first lecture was mainly about a review of the object modelling course and differences between the programming languages C and C++. Furthermore, the principle of static members and namespace could be moved to the next course, because the number of lectures would be increased by one hour. By cancelling the first lecture for a large part and some subjects, the content of the lecture could be redistributed, which created the capacity to use the debugger.

To clarify the course "Introduction OOP" by using a debugger during lectures, the DDD debugger was used to visualise objects and the Eclipse CDT debugger was used to use the tracer. More details about the explanation of the debugger choices can be found in Section 2.1.1. The DDD debugger was used to show the mutual relationships between the objects, such as the difference between fixed relationships (inheritance and composition) and loose relationships (aggregation and association). The Eclipse debugger was mainly used when tracing object-oriented statements such as creating an object. But also to show the working of a vector and the principle of a template. Working with these two debuggers was based on the graphical display of the data (DDD debugger) and a clear showing on which line in the program the debugger is located. In addition, both debuggers are free to use.

Using the debugger during the lectures. To keep the students' attention, switching between the presentation and the debugger environment had to be done without any problems. The additional problem was that the DDD debugger works in a Linux environment, and the presentations are in PowerPoint. Switching from and to the virtual machine and also within the virtual machine gave problems with the monitor. In order to avoid these problems, both the local desktop for the presentations and a laptop for the debugger environment were used. By switching from the existing desktop to the laptop, a smoother progression from presentation to the debugger was obtained

Clarifying the OOP inheritance relationship was done by displaying a visual image of the objects and the relationship between the objects and by tracking code. During the lecture, we often used different types of dogs[1] to explain the inheritance relationship, such as is shown in Figure 3.12. The class diagram shows that the class SaintBernard is derived from the class Dog which has an extra whiskey barrel. Usually, the focus was how to convert a class to C++ code, in this example how to define the class SaintBernard in the C++ definition as:

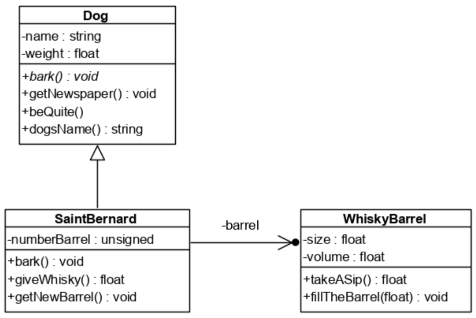


Figure 3.12: The base class Dog and the derived class SaintBernard

```
class SaintBernard : public Dog {

}
SaintBernard::SaintBernard(string n, float w,WhiskyBarrel*v) :Dog(w,n),
                              numberBarrel(0),barrel(v) {
}
```

Code fragment 3.12: Creation of a Saint Bernard object.

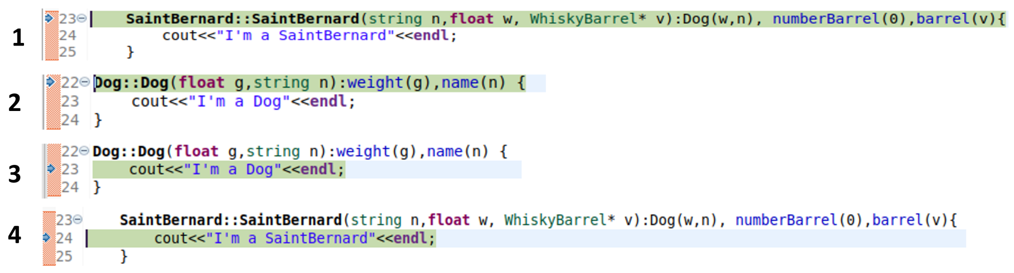
Of course the conversion from class to code was done with the necessary explanation.

With the tracer of the Eclipse debugger, it was possible to show during the lecture which steps were taken when creating an object.

```
1         WhiskyBarrel keg(1.1);
2         keg.fillTheBarrel(0.5);
3         SaintBernard sb1("Sergeant␣Dog",7.5,&keg);
```

Code fragment 3.13: Creation of a Saint Bernard object.

This was done by putting a breakpoint on line 3 of Code fragment 3.13, where the object was created, then walking through the statement with the tracer step by step. Figure 3.13 shows the steps that the



```
    SaintBernard::SaintBernard(string n,float w, WhiskyBarrel* v):Dog(w,n), numberBarrel(0),barrel(v){
        cout<<"I'm a SaintBernard"<<endl;
    }
```

```
Dog::Dog(float g,string n):weight(g),name(n) {
    cout<<"I'm a Dog"<<endl;
}
```

```
Dog::Dog(float g,string n):weight(g),name(n) {
    cout<<"I'm a Dog"<<endl;
}
```

```
    SaintBernard::SaintBernard(string n,float w, WhiskyBarrel* v):Dog(w,n), numberBarrel(0),barrel(v){
        cout<<"I'm a SaintBernard"<<endl;
    }
```

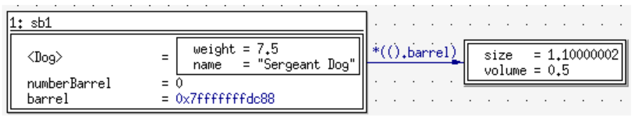
Figure 3.13: Steps to create an object of class SaintBernard

debugger does in the ”step in” mode of the tracer. In the first step, a jump to the constructor of the

[1]The idea of colleague Harry Broeders

derived class is made. In the second step, a jump to the constructor of the base class is made. In the third step, the execution of the constructor of the base class is done and finally the execution of the constructor of the derived class is done. So the base part (the dog) was created first, then the derived part was created.

By using the DDD (Data Display debugger), the mutual relationship between classes and association, during the lectures could be clarified. This could be done by using one of the essential features of the DDD debugger, i.e. visualizing the data. The debugger shows the data of the base class enclosed in the derived class and shows a pointer if a pointer was used in C++ code. When an object of the class SaintBernard was created, Figure 3.14 was generated by the DDD debugger. It shows that the base class Dog is at the top left between < > with its attributes `weight` and `name` outlined behind it. Furthermore, the attributes `numberBarrel` and the `barrel` address are displayed, and a solid arrow as a symbol that a pointer is used. In this way, we have succeeded in bringing the students a little more understanding of the relationship between the base and the derived class.



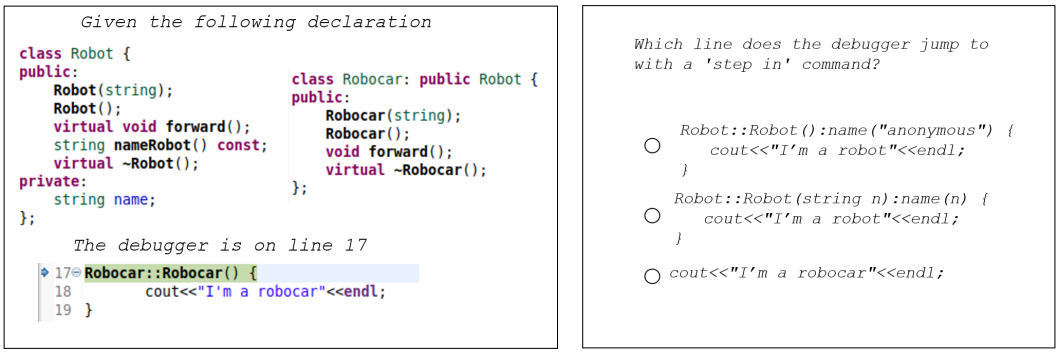
```
1: sb1
 <Dog>          = weight = 7.5
                  name   = "Sergeant Dog"
 numberBarrel   = 0
 barrel         = 0x7fffffffdc88
```

```
size   = 1.10000002
volume = 0.5
```

Figure 3.14: Object of class SaintBernard with a whiskey barrel

Using the debugger during the review of the previous lecture. To introduce the debugger to the review of the previous lecture, a non-binding automated test consist of 4 till 7 multiple-choice questions was chosen at the beginning of lectures 2 to 6 on the digital learning environment Blackboard. After the multiple-choice questions were done, they were discussed in class. For the multiple-choice questions, both types of debuggers were used.

The Eclipse debugger is the most commonly used in the multiple-choice questions. Most of the questions were in the form of ”where does the debugger go, with a ’step into’ command”. Figures



Given the following declaration

```
class Robot {
public:
    Robot(string);
    Robot();
    virtual void forward();
    string nameRobot() const;
    virtual ~Robot();
private:
    string name;
};
```

```
class Robocar: public Robot {
public:
    Robocar(string);
    Robocar();
    void forward();
    virtual ~Robocar();
};
```

The debugger is on line 17

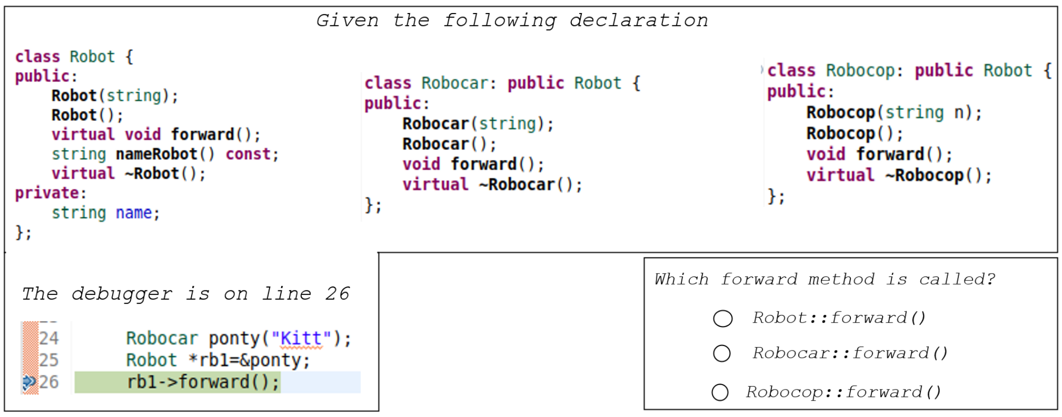
```
17 Robocar::Robocar() {
18         cout<<"I'm a robocar"<<endl;
19 }
```

Which line does the debugger jump to with a 'step in' command?

```
○ Robot::Robot():name("anonymous") {
      cout<<"I'm a robot"<<endl;
  }

○ Robot::Robot(string n):name(n) {
      cout<<"I'm a robot"<<endl;
  }

○ cout<<"I'm a robocar"<<endl;
```

Figure 3.15: An example of a multiple-choice question about constructor.

3.15 and 3.16 show two examples of multiple-choice questions that were asked via Blackboard at the beginning of the lecture. With this kind of questions, the students were tested whether they understood the topics of the previous lessons. In example of Figure 3.15, we tested whether they know that the constructor without parameters of the class Robot would be called. In example of Figure 3.16, we tested whether they know the principle of polymorphism. By allowing students to come into contact with the debugger in this way as well, we expect that the object-oriented concepts will become more transparent and that the use of the debugger will benefit more.



Given the following declaration

```
class Robot {
public:
    Robot(string);
    Robot();
    virtual void forward();
    string nameRobot() const;
    virtual ~Robot();
private:
    string name;
};
```

```
class Robocar: public Robot {
public:
    Robocar(string);
    Robocar();
    void forward();
    virtual ~Robocar();
};
```

```
class Robocop: public Robot {
public:
    Robocop(string n);
    Robocop();
    void forward();
    virtual ~Robocop();
};
```

The debugger is on line 26

```
24      Robocar ponty("Kitt");
25      Robot *rb1=&ponty;
26      rb1->forward();
```

Which forward method is called?

○ *Robot::forward()*

○ *Robocar::forward()*

○ *Robocop::forward()*

Figure 3.16: An example of a multiple-choice question about overridden a method.

Adjusting the lab sessions. The aim of the lab is to teach the students skills in embedded object-oriented programming. The environment in which the students learn their skills in the field of object-oriented programming is shown in Figure 3.17. On the laptop the assignments are developed

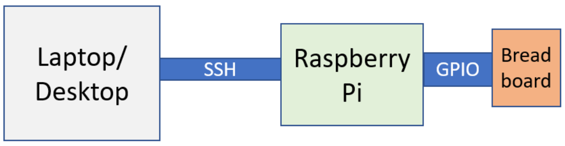


Figure 3.17: The lab envirionment.

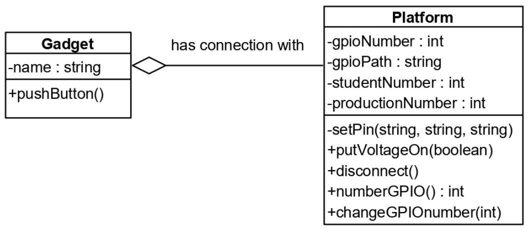
and compiled, on the Raspberry Pi the program is executed, and on the breadboard, there are simple electronic components that are read out and controlled. The students were able to download an Ubuntu VirtualBox image with the complete development environment installed (cross compiler and CDT Eclipse with working remote debugging example) on it and an image for the Rasberry Pi. The assignments that the students had to do were:

1. Remote debugging.
2. Abstract data type.
3. Inheritance and polymorphism.
4. Interface and vector.
5. Unit test.

For our study the question was: "How to ensure that the debugger will be used during the lab session?". The first assignment was an introduction to remote programming and using the GDB debugger local and remote. The students had to show local (on the Raspberry Pi) and with the remote tracer how a LED on the breadboard turned on and off. With the second, third and fourth assignment, the students had not only to demonstrate that the assignment was done but also upload screenshots to the digital learning environment (Blackboard), more detailed explanations are given in the following sections. Because the fifth assignment was an introduction to unit testing, the debugger has not been used as a tool to clarify object-oriented programming.

By including the names and study numbers in most screenshots the students had to be active with the debugger. The following paragraphs describe the modifications that were done in the case of assignments two, three and four. Assignment one and five have not been modified.

Assignment 2, data abstraction and association. The core of the second assignment was to clarify data abstraction and the difference between association and composition. The first



(a) Class diagram assignment of 2A.

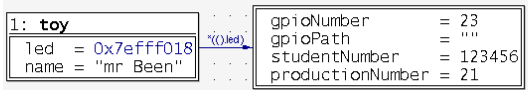
```
int main() {
//platform 23 with studie number 123456
   Platform gpio(23,123456);
   Gadget toy("mr␣Been",&gpio);
   toy.pushButton();
   sleep(1);
   toy.pushButton();
   return 0;
}
```

Code fragment (3.14) The main of assignment 2A.

Figure 3.18: The class diagram main code of assignment 2A.

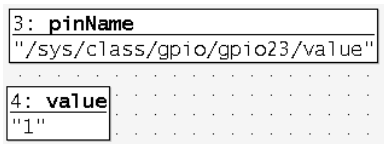
part of the assignment was an introduction to the DDD debugger. The students could download the classes **Gadget** and **Platform** as shown in Figure 3.18a and the main code as shown in Code fragment 3.14. The constructor of the class **Platform** initialized the value of the attribute `productionNumber`, which was an addition of the digits of the study number.

Using the tracer of the DDD debugger the students had to go through Code fragment 3.14 step by step and visualise the object `toy` with a pointer to the object of the class **Platform**, as shown in Figure 3.19a. Furthermore, the local variable `pinName` and `value`, which created in the method `setPin` of the class **Platform** had to be visualised, as shown in Figure 3.19b. Both figures had to upload to Blackboard. In this way, the students were forced to walk through the programme with the



```
1: toy
  led  = 0x7efff018
  name = "mr Been"

gpioNumber       = 23
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(a) Snapshot of the objects Gadget and Platform by the DDD debugger



```
3: pinName
"/sys/class/gpio/gpio23/value"

4: value
"1"
```

(b) Snapshot of a local variable by the DDD debugger

Figure 3.19: Class diagram and snapshots of assignment 2A.

help of the tracer, which introduced them to the DDD debugger and made the concepts of objects and association somewhat more evident.

The second part of the assignment consisted of implementing classes based on a class and sequence diagram. The students had to implement the classes **Duration** and **LogLed** of the class diagram, as shown in Figure 3.20. First, class **Duration** had to be implemented and tested and then reused. Second, the class **LogLed** had to be implemented using the sequence diagram, as shown in Figure 3.21.

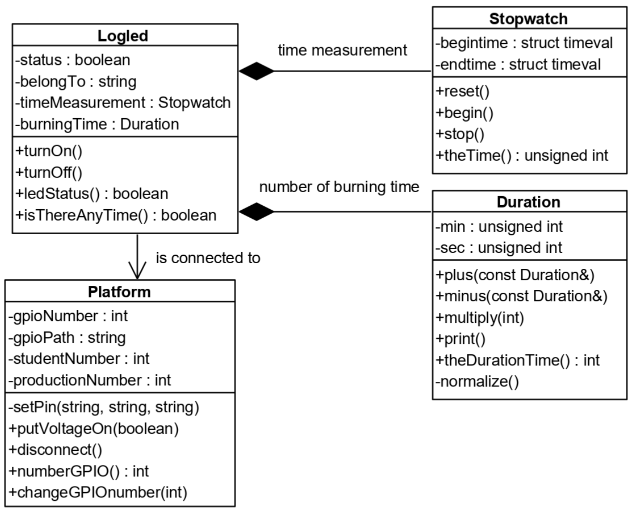


Figure 3.20: Class diagram of assignment 2B

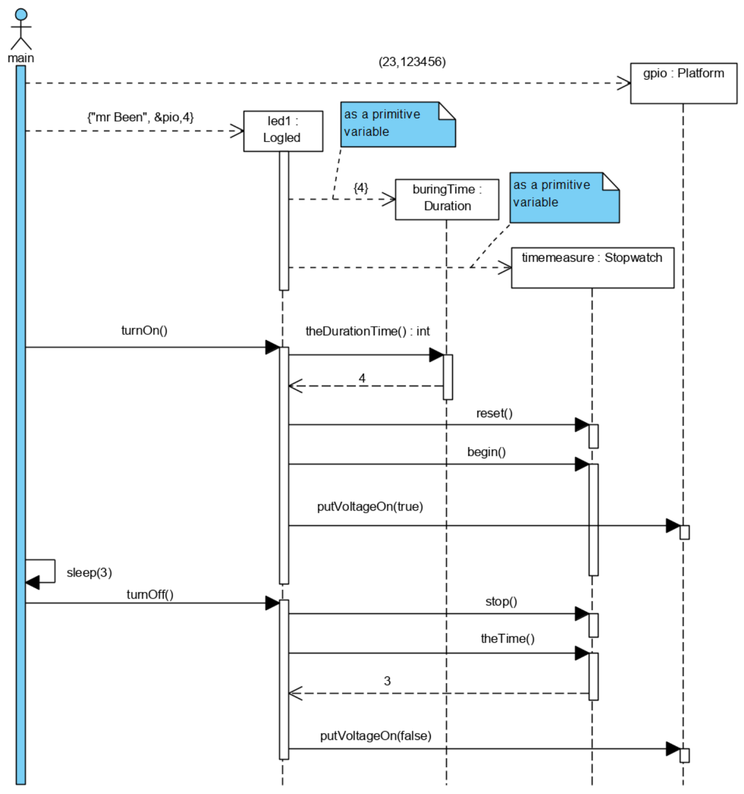
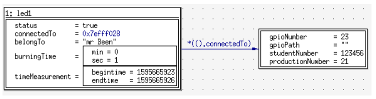


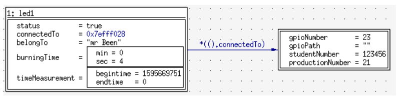
Figure 3.21: The sequence diagram of assignment 2B.

After the sequence diagram was implemented, the three screenshots had to be made, as shown in Figure 3.22. Figure 3.22a shows the value of the attributes of object led1 of the class **LogLed**. The DDD debugger shows the difference between the association and composition in C++. The composition is shown by displaying the attributes `burningtime` and `timeMeasurement` of the classes **Duration** and **Stopwatch** as a separate part within the object `led1`. While the attribute `connectionTo` is a pointer to an object of the class **Platform**. The attributes `begintime` and `endtime` are the Unix Epoch time so they should be unique. Figure 3.22b shows the situation before the method `turnOff` of the class **Logled** is called and Figure 3.22c within the method `stop` of the class **Stopwatch** where `time` is a local variable. By checking the created code with the debugger, we hypothesized that the insight into abstraction what an object is and the difference between composition and association became more evident than in previous years. By using the Epoch Unix time, the supplied screenshots



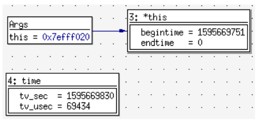
```
1: led1
status          = true
connectedTo     = 0x7efff028
belongTo        = "mr Been"
burningTime     = min = 0
                  sec = 1
timeMeasurement = begintime = 1595665923
                  endtime   = 1595665926

*(().connectedTo)

gpioNumber       = 23
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(a) Object of the class Logled at the end of the main



```
1: led1
status          = true
connectedTo     = 0x7efff028
belongTo        = "mr Been"
burningTime     = min = 0
                  sec = 4
timeMeasurement = begintime = 1595669751
                  endtime   = 0

*(().connectedTo)

gpioNumber       = 23
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(b) Logled object after calling method turnOn.



```
Args
this = 0x7efff020

3: *this
begintime = 1595669751
endtime   = 0

4: time
tv_sec  = 1595669830
tv_usec = 69434
```

(c) L ocal variable time in the method stop of the class Stopwatch.

Figure 3.22: Snapshots of assignment 2B.

would be unique.

Assignment 3, inheritance and dynamic binding. Assignment three of the practical was primarily about inheritance and polymorphism. Just like assignment two, the assignment was made around a class diagram, which is shown in Figure 3.23 and the main code, which is shown in Code fragment 3.15. The students had to implement the classes **Led**, **SingleLed** and **DualLed**, then the classes had to be tested using Code fragment 3.15. After the code could be executed, with

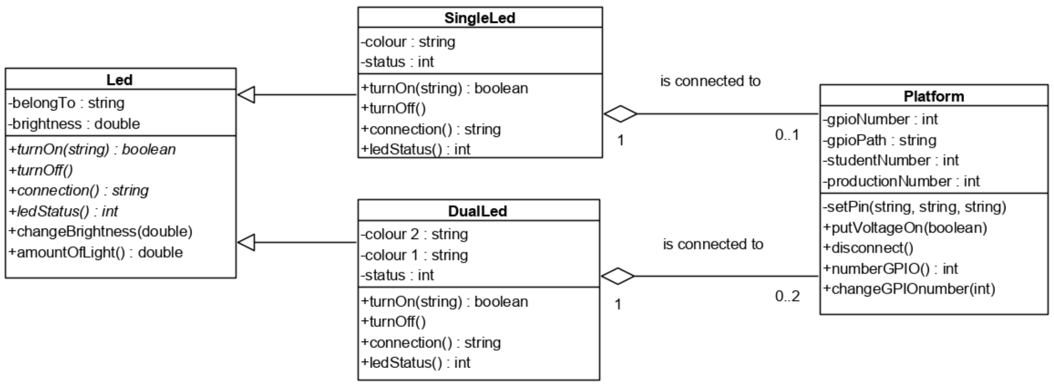


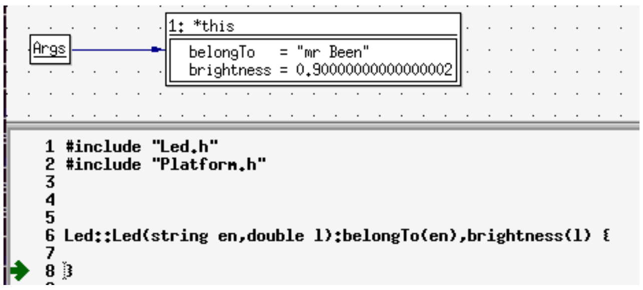
Figure 3.23: The class diagram of assignment three.

the DDD debugger the programme was walked through, and screenshots of visualised objects had to be taken regularly. By taking screenshots, the correct functioning of the programme could be demonstrated.The assignment is divided into three parts. The first part is about creating a derived class, the second part about overloading the copy constructor and the last part about dynamic binding and polymorphism.

```
1   int main() {
2     Platform gpioA(18,123456); //platform 18 with study number 123456
3     SingleLed sl1("mr␣Been",0.9,"green",&gpioA);//name=mr Been, brightness=0.9 lumen
4     SingleLed led2(sl1);  //copy constructor
5     sl1.changeBrightness(1.2);
6     sl1.turnOn("groen");
7     sleep(1);
8     led2.turnOff();
9     return 0;
10  }
```

Code fragment 3.15: The first main of assignment three

Clarifying the principle of the derived class, the students had to walk with the tracer through the constructor step by step and take corresponding screenshots. First, place a breakpoint on line 3 of

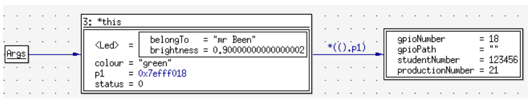


```
1: *this
belongTo   = "mr Been"
brightness = 0.90000000000000002
```

```
#include "Led.h"
#include "Platform.h"



Led::Led(string en,double l):belongTo(en),brightness(l) {

}
```

Figure 3.24: During the creation of the base class.

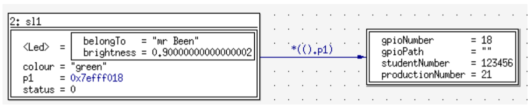
Code fragment 3.15 and jump into the constructor of the class **SingleLed**. Second, a screenshot had to be taken, when the tracer was inside the base class of a Singleled, as is shown in Figure 3.24. Third, a screenshot had to be taken, at the end of the initialisation of the derived class, as is shown in Figure 3.25a. And finally, a screenshot had to be taken, after object `sl1` is created, as is shown in Figure 3.25b. In this way, the students used the debugger, and we hypothesised that it would



```
3: *this
<Led>  = belongTo   = "mr Been"
         brightness = 0.90000000000000002
colour = "green"
p1     = 0x7efff018
status = 0
```

```
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(a) During the creation of the derived class.



```
2: sl1
<Led>  = belongTo   = "mr Been"
         brightness = 0.90000000000000002
colour = "green"
p1     = 0x7efff018
status = 0
```

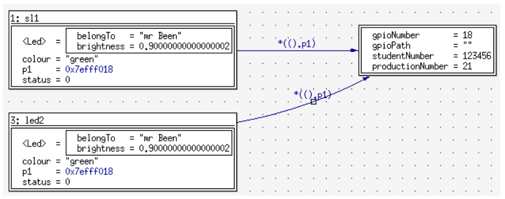
```
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(b) After the object is created.

Figure 3.25: Snapshots during the creation of an object of class SingleLed.

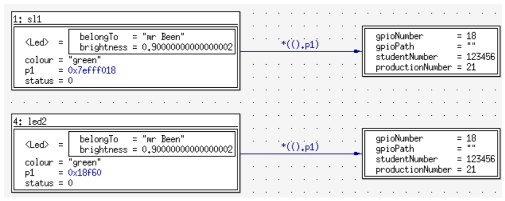
be more evident for the students that the base class is part of the derived class. Checking if the students understood the copy constructor, a screenshot would have to take of the original object and the copied object. The copy constructor is called by creating object led2 on line 4 of Code fragment 3.15. By visualizing both objects `sl1` and `led2` after line 4 is executed, the distinction whether the copy constructor is overloaded or not, is shown in Figure 3.26. By visualising both objects, we expected that it became more noticeable when a copy constructor should be used.

The last part of this assignment was to clarify dynamic binding in involving classes with overidden methods. During the lectures, it had already been demonstrated that by overriding the method of



```
1: sl1
<Led> = belongTo   = "mr Been"
        brightness = 0.9000000000000002
colour = "green"
p1     = 0x7efff018
status = 0

3: led2
<Led> = belongTo   = "mr Been"
        brightness = 0.9000000000000002
colour = "green"
p1     = 0x7efff018
status = 0

*(().p1)
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(a) A copy without copy constructor.

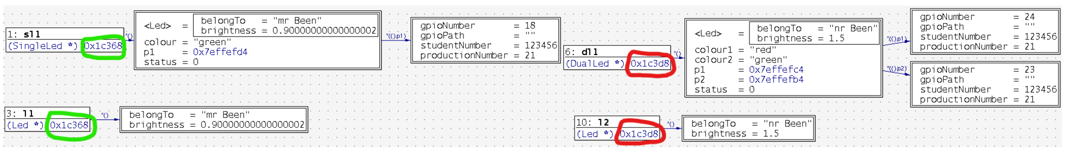


```
1: sl1
<Led> = belongTo   = "mr Been"
        brightness = 0.9000000000000002
colour = "green"
p1     = 0x7efff018
status = 0

*(().p1)
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21

4: led2
<Led> = belongTo   = "mr Been"
        brightness = 0.9000000000000002
colour = "green"
p1     = 0x18f60
status = 0

*(().p1)
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

(b) A copy with copy constructor.

Figure 3.26: Snapshots of copy from a singleLed without and with copy constructor

the derived class is invoked. To increase the students' understanding of dynamic binding and which parts of the object are visible, the students had to take screenshots, of a base and derived pointer which both refers to the same object as can be seen at the addresses in the Figure 3.27. The objects



```
1: sl1
(SingleLed *) 0x1c368
*()
<Led> = belongTo   = "mr Been"
        brightness = 0.9000000000000002
colour = "green"
p1     = 0x7effefd4
status = 0
*(().p1)
gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21

3: l1
(Led *) 0x1c368
*()
belongTo   = "mr Been"
brightness = 0.9000000000000002

6: dl1
(DualLed *) 0x1c3d8
*()
<Led>   = belongTo   = "nr Been"
          brightness = 1.5
colour1 = "red"
colour2 = "green"
p1      = 0x7effefc4
p2      = 0x7effefb4
status  = 0
*(().p1)
gpioNumber       = 24
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
*(().p2)
gpioNumber       = 23
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21

10: l2
(Led *) 0x1c3d8
*()
belongTo   = "nr Been"
brightness = 1.5
```

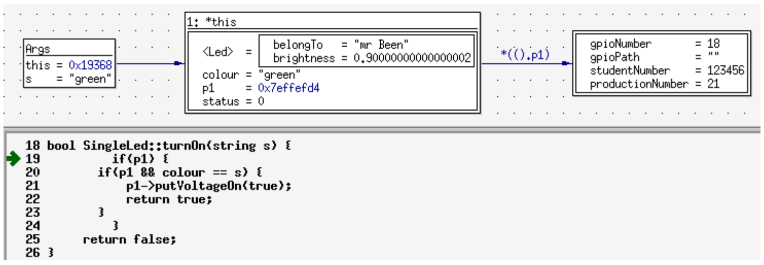
Figure 3.27: A base and derived pointer refers to the same object.

were created using Code fragment 3.16. The students had to take a screenshot of a base and derived pointer to objects of singleled and dualled. Figure 3.27 shows that pointer `sl1` refers to an object of the class **SingleLed** that is a derivative of the class **Led** and is located at address 0x19368. With the statement "`Led* l1=sl1;`", a pointer is created of the base class Led that refers to the same object which `sl1` refers to. Figure 3.27 shows that both pointers refer to the same address and the derived part is hidden for `l1`.

```
Platform gpioA(18,123456);
Platform gpioB(24,123456);
Platform gpioC(23,123456);

//belongs to="mr Been" brightness= 0.9 lumen
SingleLed* sl1=new SingleLed("mr␣Been",0.9,"green",&gpioA);

//belongs to ="mr Been" brightness= 1.5 lumen
DualLed* dl1=new DualLed("nr␣Been",1.5,"red","green",&gpioB,&gpioC);
Led* l1=sl1;
Led* l2=dl1;

l1->turnOn("green");
sleep(1);
l2->turnOn("green");
sleep(1);
l2->turnOn("red");
sleep(1);

Led* uit=sl1;
uit->turnOff();
sleep(1);
uit=dl1;
uit -> turnOff();
```

Code fragment 3.16: Derived and base pointer refer to the same object.

The actual demonstration that dynamic binding has been used was done on lines 10 and 12 of Code fragment 3.16 by calling the function turnOn of the derived class with a pointer of type Led. The ”step” function of the DDD debugger was used to enter the turnOn function. The code clearly shows that the derivative method is called. This is confirmed by making the object visible, as shown in Figure 3.28.



```
Args
this = 0x19368
s    = "green"

1: *this
<Led> =   belongTo   = "mr Been"
          brightness = 0.9000000000000002
colour = "green"
p1     = 0x7effefd4
status = 0

*(().p1)

gpioNumber       = 18
gpioPath         = ""
studentNumber    = 123456
productionNumber = 21
```

```
bool SingleLed::turnOn(string s) {
        if(p1) {
      if(p1 && colour == s) {
          p1->putVoltageOn(true);
          return true;
      }
        }
    return false;
}
```

Figure 3.28: Calling the method, which is overridden.

Assignment 4, one-to-many relationship. The fourth assignment aimed to become acquainted with a one-to-many relationship and to create independence by using an interface. For the one-to-many relations, a vector is used, and for making objects independent [39], the principle of interface type is used. For this assignment, the class diagram as shows in Figure 3.29 is used. The students had to implement the class **Ledlamp**, the interface **ConnectionPlatform**, and the class **Raspberry**, which looks like the class **Platform** from the previous assignments. In contrast to the

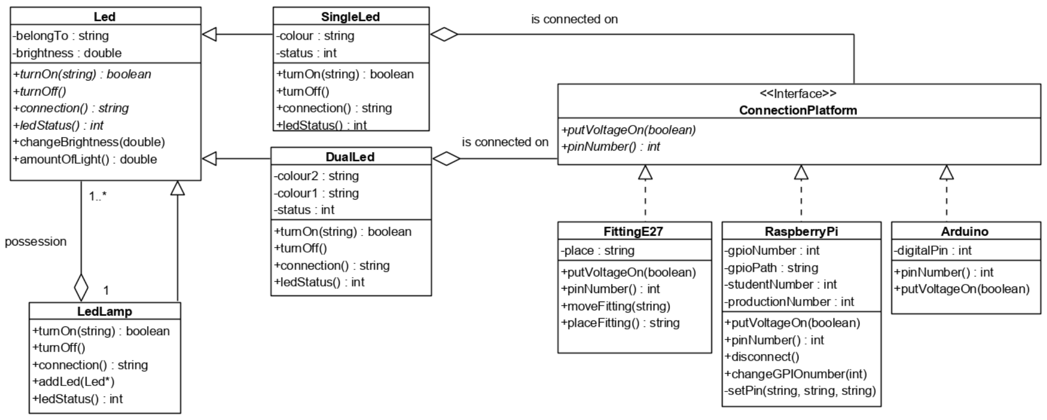


Figure 3.29: Class diagram belongs to assignment 4.

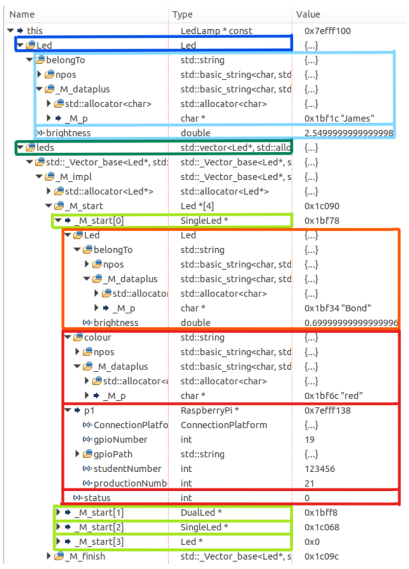
previous assignment, this assignment used the Eclipse environment so that students could also see the content of the objects in a different way.

Using Code fragment 3.17, the students had to use the ”step in” of the tracer to call the method `turnOn` of ledlamp `l1` (line 13) and make the variable visible, so that a view looks like Figure 3.30b was created, which they had to upload without contours. Figure 3.30b, shows differently than the DDD debugger that the base class **Led** is a part of the derived class **LedLamp**; this can be seen under `this` (the ledlamp) but also in the vector. With a non-visual debugger, it can also be shown that a base class is part of a derived class, although this is not as clear as with visual debuggers such as the DDD debugger, but a non-visual debugger is more often integrated with an IDE like Eclipse and Visual Studio.

```
1  //gpio port 12, study number 123456
2  RaspberryPi connection2(12,123456);
3  RaspberryPi connection3(13,123456);
4  RaspberryPi connection4(19,123456);
5  RaspberryPi connection5(24,123456);
6  LedLamp ll("James");
7  ll.addLed(new SingleLed("Bond",0.7,"red",
8                      &connection4));
9  ll.addLed(new DualLed("007",1.2,"red","green",
10                     &connection2,&connection3));
11 ll.addLed(new SingleLed("Sean",0.65,"green",
12                     &connection5));
13 ll.turnOn("red");
14 sleep(3);
15 ll.turnOn("green");
16 sleep(3);
17 ll.turnOff();
18 sleep(1);
```

Code fragment (3.17) main code of assignment 4.

(a) Code to get Figure 3.30b.



```
Name                          Type                            Value
▾ this                        LedLamp * const                 0x7efff100
  ▾ Led                       Led                             {...}
    ▾ belongTo                std::string                     {...}
      ▸ npos                  std::basic_string<char, std     {...}
      ▾ _M_dataplus           std::basic_string<char, std     {...}
        ▸ std::allocator<char>  std::allocator<char>          {...}
        ▸ _M_p                char *                          0x1bf1c "James"
    brightness                double                          2.5499999999999998
  ▾ leds                      std::vector<Led*, std::allo     {...}
    ▾ std::_Vector_base<Led*, std:  std::_Vector_base<Led*, s {...}
      ▾ _M_impl               std::_Vector_base<Led*, s       {...}
        ▸ std::allocator<Led*>  std::allocator<Led*>          {...}
        ▾ _M_start            Led *[4]                        0x1c090
          ▾ _M_start[0]       SingleLed *                     0x1bf78
            ▾ Led             Led                             {...}
              ▾ belongTo      std::string                     {...}
                ▸ npos        std::basic_string<char, std     {...}
                ▾ _M_dataplus std::basic_string<char, std     {...}
                  ▸ std::allocato  std::allocator<char>       {...}
                  ▸ _M_p      char *                          0x1bf34 "Bond"
              brightness      double                          0.69999999999999996
            ▾ colour          std::string                     {...}
              ▸ npos          std::basic_string<char, std     {...}
              ▾ _M_dataplus   std::basic_string<char, std     {...}
                ▸ std::allocator<  std::allocator<char>       {...}
                ▸ _M_p        char *                          0x1bf6c "red"
            ▾ p1              RaspberryPi *                   0x7efff138
              ConnectionPlatfo  ConnectionPlatform            {...}
              gpioNumber      int                             19
              ▸ gpioPath      std::string                     {...}
              studentNumber   int                             123456
              productionNumb  int                             21
            status            int                             0
          ▸ _M_start[1]       DualLed *                       0x1bff8
          ▸ _M_start[2]       SingleLed *                     0x1c068
          ▸ _M_start[3]       Led *                           0x0
        ▸ _M_finish           std::_Vector_base<Led*, s       0x1c09c
```

(b) Debug variable during calling method turnOff (line 12).

Figure 3.30: Code and the content of the object ll with a vector of different types of leds.

3.4.3 Adjustments the course object-oriented programming part II.

In contrast to the course Introduction object-oriented programming, this course did not have capacity problems with the lectures. That is because the number of lectures has extended from one to two lecture hours per week. Because of the topics, this course is more applied and working with objects instead of understanding of objects. The core of this course were expansion of the course Introduction object-oriented programming such as the STL data-structures, exceptions and C++11 features.

In this course, we have chosen to concentrate on the use of the debugger more in the lectures. The main reasons for doing it this way were:

- The basic principles of classes, objects and the relationships between the objects have already been discussed in detail. The subjects that were treated lend themselves less to visualisation.
- Because screenshots would have to be retaken, from a debugging environment, there is a realistic chance that the students would become demotivated.

By not changing the lab sessions any further, it remained clear to the students that the focus was on object-oriented programming and not on working with a debugger.

In order to use the debugger to clarify the OOP topics, we decided to split the lectures into three parts: First, this is the presentation to explain new topics using the debugger. This part is similar to the previous course, only for less time and topics. Second, students had to do a short assignment during the second part of the lectures and one or two students should be explained to the other students with the help of the debugger. Finally, to review the previous college, which was done based on a small assignment that the students had to do at home.

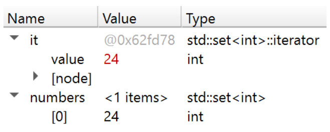
Using the debugger during the presentations. In contrast to the course Introduction object-oriented programming, during the lectures of this course, we did not work with the DDD debugger. All the examples were done with QT Creator. The students could choose their favourite IDE during the part that a small assignment had to be done.

One aspect that could be clarified with the debugger during the lecture is the fact that the STL handles its memory management and the possible problems that may arise when using iterators. If an element is erased or the memory location has been changed, the iterator becomes invalid [46]. This phenomenon could possible be clarified with the help of the debugger. When we go step-by-step through Code fragment 3.18 with the debugger and watch the contents of the set and the iterator, Figure 3.31a becomes visible after the execution of line 6. Here we see that the set only contains the element 24 and the iterator refers to this element.

```
1   int main()
2   {
3           set<int> numbers;
4           set<int>::iterator it;
5           numbers.insert(24);
6           it=numbers.begin();
7           numbers.insert(17);
8           numbers.insert(20);
9
10          numbers.erase(24);
11          while(it != numbers.end()) {
12                  cout<<*it;
13                  it++;
14          }
15          return 0;
16  }
```

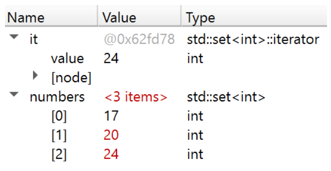
Code fragment 3.18: Problem with the iterator and memory management of the set container.

If elements 17 and 20 are added, Figure 3.31b shows that the elements are sorted, but that the iterator no longer refers to the beginning, but still refers to element 24. If element 24 is removed, Figure 3.31c shows that element 24 has been removed from the set, but that the iterator has become invalid. With the combination of executing step-by-step and watching variables with the debugger, it should



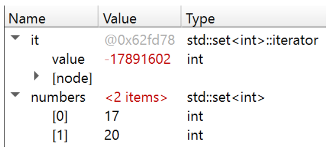
```
Name          Value        Type
▾ it          @0x62fd78    std::set<int>::iterator
    value     24           int
  ▸ [node]
▾ numbers     <1 items>    std::set<int>
    [0]       24           int
```

(a) After executing of line 6 in Code fragment 3.19.



```
Name          Value        Type
▾ it          @0x62fd78    std::set<int>::iterator
    value     24           int
  ▸ [node]
▾ numbers     <3 items>    std::set<int>
    [0]       17           int
    [1]       20           int
    [2]       24           int
```

(b) After executing of line 8 in Code fragment 3.19.



```
Name          Value        Type
▾ it          @0x62fd78    std::set<int>::iterator
    value     -17891602    int
  ▸ [node]
▾ numbers     <2 items>    std::set<int>
    [0]       17           int
    [1]       20           int
```

(c) After executing of line 10 in Code fragment 3.19.

Figure 3.31: The content of the STL container set, in Code fragment 3.19.

be possible to clarify some run-time errors that could occur by using iterators and the STL container.

Doing a short assignment during the lectures. After the presentation part of the lecture, the students had to do a short assignment and one or two students had to demonstrate their solution by using the debugger. In order to emphasize the subjects covered, the students could download the project from QTCreator or Visual Studio IDE[2] from the digital learning environment Blackboard.

[2]Many students wanted to use Visual Studio

An example how to use a set is shown in Code fragment 3.19. The students could download the project or copy the main code from the sheets. In this example, they only have to concentrate about working with a set.

```
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main()
{
  set<char> guessed_letter;
  set<char>::iterator occurs;
  pair<set<char>::iterator,bool> inTheSet;
  string word;
  char letter;
  word="examination";
  bool finish=false;
    do {
          string::iterator bookmark;
          finish=true;
          for(bookmark=word.begin();bookmark !=word.end();bookmark++) {
                 //loop  through the set of guess words.
                 //and print '.' or letter
          }
          if(!finish) {
                cout<<endl<<"guess␣a␣letter:"; cin>>letter;
                // place letter in the set and check if letter already is set.
          }
        }while(!finish);
        cout <<endl<<"You␣have␣guessed␣the␣word"<<endl;
        return 0;
}
```

Code fragment 3.19: Problem with the iterator and memory management of the set container.

In this way, practical skills could be acquired, and by involving the debugger, several possibilities of the set were also made visible.

3.4.4 Conclusion regarding adjusting education material.

It is quite possible to use the debugger differently during both OOP courses to clarify the concepts of object-oriented programming. By distinguishing between the two courses when using the debugger, it was possible to prevent that students saw the debugger as a burden rather than as a tool.

3.5 Conclusion regarding experimental setup

The analysis of both examinations indicated that students had problems with several object-oriented programming concepts in the C++ programming language. These concepts could probably be clarified by using the debugger during lectures and lab sessions to visualise objects and by going through a combination of a sample programme step by step and making the changed content of the variables visible.

Chapter 4

Results and analysis of the experiments

This chapter discusses the results of the exams of both courses and the results of the questionnaire which the students completed in the last week of the course Introduction object-oriented programming.

4.1 Analysis of the current ”Introduction object-oriented programming” exam

The written exam was the leading supplier of data to determine whether the hypothesis ”People who use a debugger when learning object-orient programming better understand the concepts” is accepted or rejected. The exam is compared with the results of two related courses, as described in Section 3.1.4, which were taught before this course to the same cohort. Furthermore, we compare the results of this edition of the course with the two previous editions. The second part of this section discusses the results of the individual questions and compares them with the questions analysed in Section 3.2.

4.1.1 The results compared to previous years and other courses

In order to determine the extent to which the students' results deviate from the results of previous years, the result of the exam will be compared with two previously taken exam in the same field. These are the courses Object-oriented modelling (OOMOD) that was taught a term earlier and Embedded programming (EMBED) that was taught in the third term of the foundation year. A complete overview of the courses in the same field is shown in Figure 3.1. The students who took part in the exam can roughly be divided into the following three groups:

- The students who followed the course for the first time.
- The students who had already taken the course at least once before and attended the adapted lectures.
- The students who only participated in the examination.

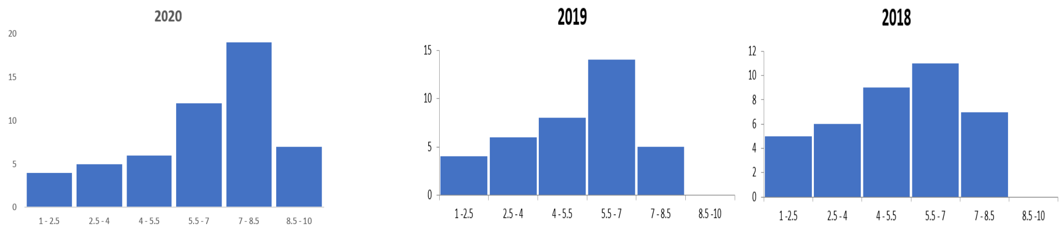
To try to get an accurate view of the extent to which the group of students performed better or worse than previous years. An analysis was also done of students who took exams without delay in the EMBED, OOMOD and OOPR1 courses.

Comparing with all the participants. The results of the OOPR1, OOMOD and EMBED[1] exams of the past three years, in which all participants are counted, are shown in Table 4.1. Besides

academic year		OOPR1	OOMOD	EMBED
2019 - 2020	number of participants	53	49	64
	success rate	71.7%	59.2%	48.4%
	average	6.45	5.3	4.9
	variance	4,65	1.2	5.3
2018 - 2019	number of participants	37	44	56
	success rate	51.4%	70.5%	62.5%
	average	5.2	6.2	5.8
	variance	3.6	2.2	3.1
2017 - 2018	number of participants	38	51	71
	success rate	50.0%	62.7%	52.1%
	average	5.1	5.8	4.9
	variance	3.7	2.2	6.0

Table 4.1: Results of the exams with all the participants in order of execution (OOPR1 is last).

the results of the course Introduction object-oriented programming, the results of two related courses that were previously taken at the same cohort are also included. To get insight into the distributions of OOPR1 results before and after the course has been modified. We have put the results of each OOPR1 course in a histogram, which are shown in Figure 4.1. This shows that the OOPR1 exam, after applying the debugger during the lessons, has improved compared to the previous two years.



(a) Exam academic year 2019. (b) Exam academic year 2018. (c) Exam academic year 2017.

Figure 4.1: The distribution of the results of the exams Introduction object-orieted programming.

To be able to get an indication whether the adjustments done to the OOPR1 course are statistically significance, we used the Mann-Whitney U Test[2]. The Mann-Whitney U Test was taken because the data is not parametric and the histograms of Figure 4.1 show that it is not a normal distribution. Table 4.2 shows the statistical significance (p-value), Z and the Mann-Whitney U value, which are calculated by SPSS. Furthermore, the rank-biserial correlation, which is a correlation effect size for the Mann-Whitney U test, is calculated with Hans Wendt's formula ($1 - \frac{2U}{n_1 n_2}$) [33, 56] and is always between zero and one. Another way to calculate the effect size can be done following formula $r = \frac{Z}{\sqrt{n_1+n_2}}$ [47].

[1]This course is from the foundation year, so it is in the previous academic year

[2]https://wikistatistiek.amc.nl/index.php/KEUZE_TOETS#Ordinale_variabelen

OOPR1		calculation by SPSS			Rank-Biserial Correlation (r)
comparing academic years		Mann-Whitney U	Z	Exact Sig. (2-tailed) (p-value)	
2019	2018	620	-2.957	0.03	0.37
2019	2017	595	-3.317	0.01	0.41

Table 4.2: Statistic results of between the exams OOPR1.

When we compare the results of the 2019 exam (after the intensive use of the debugger) with the results of the exams for the adjustments, we see that both the success rate and the average have improved. Table 4.1's results show that the success rate increased from 50% in 2017 and 51.4% in 2018 to 71.7% in 2019. We also see that the average had increased from 5.1 in 2017 and 5.2 in 2018 to 6.5 in 2019. Table 4.2 shows that the 2019 examination is statistically significant with the 2017 and 2018 examinations where the p-value is below 0.05[3]. Furthermore, the effect size is between 0.30 and 0.5, which indicates that the effect is moderate[47]. The p-value and effect size indicate that the improved results do not seem to be based on coincidence. It is also essential to know whether the 2019 group does not perform better with other comparable courses.

To exclude that students in the academic year 2019-2020 were better than in the two previous years, we also analyse two related courses, which were previously taught the same cohort. When we analyse the results of the two related courses, we see a deterioration instead of an improvement between the results of the last exam and the exams of the previous two years. For the OOMOD course, the success rate decreased from 70.5% to 59.2% and the average from 6.2 to 5.3. The p-value calculated according to the Mann-Whitney method is 0.03 and the effect size is 0.36. This indicates that there would be a statistic significant with a medium effect size. The reason could be that the lectures in the academic year 2019 were taught by another non-experienced lecturer who reused the sheets. For the EMBED course, the success rate fell from 62.5% to 48.4% and the average from 5.8 to 4.9. The p-value here is 0.72 and the effect size is 0.19. This indicates that there would be no statistic significant with a low effect size. The students who did the OOPR1 exam after the course had changed, did not perform better on the related courses of the past two years. The disadvantage of Table 4.1 is that there are lots of participants who did not the exams for the first time.

Comparing with participants who also did the other courses. When we only analyse the participants who participated in all three exams and did them in the right order, first EMBED, then OOMOD and finally OOPR1, the course OOPR1 still did better last year. Table 4.3 shows the results.

Sixty-six per cent of the participants in the ”OOPR1” exam of the academic year 2019 had also taken part in the OOMOD and EMBED exams of the regular study programme. This indicates that one-third of the participants in OOPR1 did not participate in the exams according to the regular programme. These may be participants who had taken the exam earlier or were absent at the OOMOD or EMBED exam. Furthermore, the course EMBED results much better in Table 4.3 than in Table 4.1; this has to do with the students that left the study program after the foundation year. When we filter out the students who did not do the standard order of the courses EMBED, OOMOD and OOPR1, we do not observe any fundamental changes over the results of the last OOPR1.

Based on the results of the exam of Introduction object-oriented programming, it could be concluded that using the debugger to clarify object-oriented programming was successful. In order to

[3]https://wiki.uva.nl/methodologiewinkel/index.php/Mann-Whitney_test

academic year	number of participants		OOPR1	OOMOD	EMBED
2019 - 2020	35	success rate	65.7%	68.6%	65.7%
		average	6.21	5.63	5.96
		variance	5.12	10.91	3.08
2018 - 2019	22	success rate	59.1%	81.8%	81.8%
		average	5.6	6.8	6.3
		variance	3.4	2.2	2.3
2017 - 2018	31	success rate	41.9%	62.29%	78.1%
		average	4.8	5.7	5.8
		variance	3.3	2.0	4.1

Table 4.3: Results of the exams with participants who participated in all three examinations.

find out which parts were successful and which were not, the next part analyses the exam with the similar questions as discussed in Section 3.2.1.

4.1.2 Results per question

The Introduction object-oriented programming exam has been done well after applying the debugger, as can be read in the previous section. In this section, the individual questions are analysed and compared with similar questions from Section 3.2 (Analysis of the exam Introduction object-oriented programming, January 2019). The class diagram, which is important part to understand the question is shown in Figure 4.2. The complete exam can be found in Appendix C.

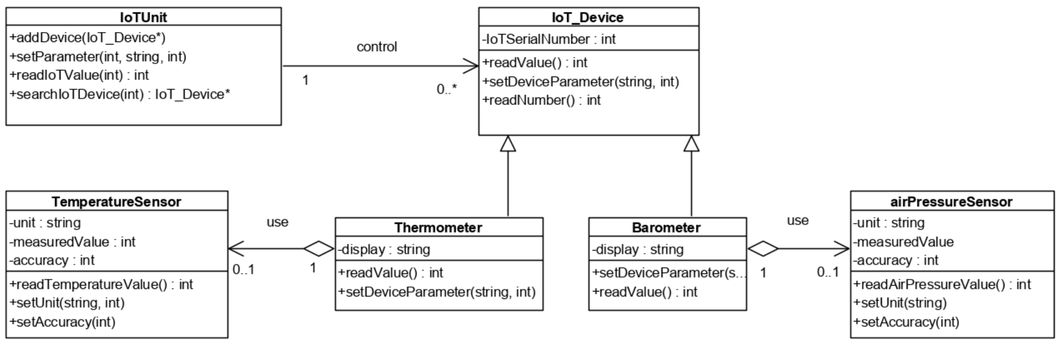


Figure 4.2: Class diagram of assignment 1 of the exam Introduction object-oriented programming.

Question A: What is the consequence in this example if `virtual` of the method `virtual double readValue(int);` of the class **IoT_Device** is omitted.

This question tested the knowledge about dynamic binding between the base and derived class by using the keyword `virtual`. The results of this question and the similar question from the exam in 2019 are shown in Table 4.4. These results show that the question was better answered. The average score increased from 1.08 to 2.49, and the variance increased to 5.02, which indicates that a lot of students got the question right or wrong. The R_{it} has been increased from 0.24 to 0.46, which indicates that the question was more distinctive than the one in 2019. Despite using the debugger

	max points	average	variance	M/Mmax	R_{it} exam
January 2019	5	1.08	4.02	0.22	0.24
January 2020	5	2.49	5.02	0.5	0.46

Table 4.4: Result of question A.

deeply, the M/Mmax is still lower then 0.5, but using the debugger seems to have a positive effect on the concept of understanding the keyword `virtual`.

Question B: What is the consequence in this example if *=0* after the method `virtual double readValue( int ) const;` of the class **IoT_Device** is placed.

This question tested knowledge about the pure virtual method and the class abstraction by using the statement *=0*. The results of this question and the similar question from the exam in 2019 are shown in Table 4.5. In contrast to question A, this question does not see a clear improvement of the

	max points	average	variance	M/Mmax	R_{it} exam
January 2019	5	2.32	2.89	0.46	0.40
January 2020	5	2.68	2.8	0.54	0.68

Table 4.5: Result of question B.

answers. The average increased from 2.32 to 2.68, the variance has remained more or less the same, and the question was more distinct than the one in 2019. Although the use of the debugger did not play an essential role in this question, only indirectly in the slicing problem (the loss of attributes), the results in this question show a better result.

Question G: The method `double readIoTValue (int) const;` of the class **IoTUnit** returns the value of the platform that is located on the given position and the pin in question. Write down the implementation (program code) of the method:
`double readIoTValue (string) const;`

This question tested the knowledge about looping through the vector and calling other methods of another class. The results of this question and the similar question from the exam in 2019 are shown in Table 4.6. It is noticeable that the M/Mmax rises from 0.47 to 0.61. This means that

	max points	average	variance	M/Mmax	R_{it} exam
January 2019	10	4.7	15.8	0.47	0.74
January 2020	10	6.13	14.39	0.61	0.79

Table 4.6: Result of question G.

61% of the maximum number of points has been achieved, which is a considerable improvement. Furthermore, there is still a broad spread of results (the variance has decreased slightly but still has a high value of 14.39), and the distinctiveness of the question (R_{it}) has become even slightly higher. Working with the debugger might have helped a higher percentage of students to better understand the subjects ”looping through the vector” and ”calling other methods of another class”. However, a higher spread indicates that a fair portion of students has not understood these subjects.

When we look at the M/Mmax of the different parts of question G, we see the M/Mmax results per part in Table 4.7. The parts related to ”looping through the vector” and ”calling the method

	total	looping though the vector	calling method I (readNumber)	calling method II (readValue)	remainder diverse
January 2019	0.47	0.47	0.41	0.59	0.38
January 2020	0.61	0.73	0.63	0.69	0.36

Table 4.7: The mean/Max per parts of question G.

`readNumber`" have improved the most. During the lab sessions, the students had to walk with the tracer to the right position so that they could take the desired screenshot. Working with the debugger has probably made a positive contribution to improving the results of this question.

Question H: Give the implementation (program code) of the constructor of the class **Barometer**.

This question tested the knowledge about the constructor of a derived class where the constructor of the base class had to be called. The students did this question better in 2020 than the similar question in 2019. The M/Mmax increased from 0.40 to 0.78 while the R_{it} remained at the same high level (0.74 in 2019 and 0.76 in 2020), which indicates that the question has maintained its distinctiveness. Table 4.8 shows the results of this question in 2019 and 2020. When we look at

	max points	average	variance	M/Mmax	R_{it} exam
January 2019	10	4.38	10.58	0.44	0.74
January 2020	10	7.79	10.63	0.78	0.76

Table 4.8: Result of question H.

the M/Mmax of the different parts, we see the M/Mmax results per part in Table 4.9. The students understood better how calling the base class constructor inside the constructor of the derived class works. Both during multiple-choice questions at the beginning of the lecture and as a demonstration

	calling base constructor	initialisation attribute I (dislay)	initialisation atribute II (air pressure sensor)
January 2019	0.39	0.84	0.25
January 2020	0.74	0.90	0.77

Table 4.9: The mean/Max per parts of question H.

during the lecture, we often discussed by using the debugger. That the initialisation of attribute II was done better can be explained because attribute II of the exam had to be initialised to 0 in 2019, while attribute II of the exam in 2020 was a constructor parameter. Using the debugger could be useful to explain the constructor of the derived class.

Question I: The method *`void setDeviceParameter(string w, int value);`* of the class **Barometer**, if the sensor is present, set the unit and accuracy of the sensor. If no sensor is present, nothing happens. Write down the implementation (program code) of the method:
`void setDeviceParameter(string w, int value);`

This question corresponds to question J of the 2019 exam and tested a 0 .. 1 association combined with the invocation of two different methods. Although a comparable question in 2019 was not poorly done, the students did this one better than the similar question in 2019. The results of both questions are shown in Table 4.10. This question combines a high M/Mmax value with a high

	max points	average	variance	M/Mmax	R_{it} exam
January 2019	10	6.68	12.00	0.67	0.73
January 2020	10	7.17	14.53	0.72	0.86

Table 4.10: Result of questing I.

R_{it} value, which indicates that the question is relatively simple, but it is still distinctiveness with a R_{it} of 0.86.

When we look at the M/Mmax of the different parts, we see the M/Mmax results per part in Table 4.11. The M/Mmax of the ’null pointer’ control improved the most from 0.66 to 0.77, and the two method invocations both slightly improved corresponding to the invocation of method II of question G. The use of the debugger could have done a small bit to improve the result of this

	check ’NULL pointer’	method I (configure)	method II (set-Measurement)
January 2019	0.55	0.67	0.65
January 2020	0.77	0.72	0.70

Table 4.11: The mean/Max per parts of question I.

question.

Question 2B: This question used another class diagram, which is shown in Figure 4.3. The question was:
Implement the copy constructor of the class **Torch**, so that with the statement *`Torch reserveLight(light);`* the reserveLight object is the similar to the object light but with different LEDs and batteries.

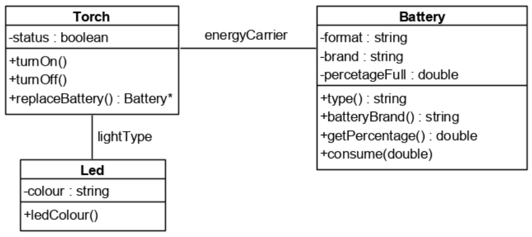


Figure 4.3: Class diagram of assignment 2.

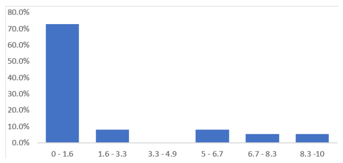
Because the subject copy constructor question was usually poorly made, we decided to use the same question as the question of the 2017 exam, to increase the accuracy of the measurement, whether the use of the debugger was successful. The results of both questions and the comparable question of the 2019 exam are shown in Table 4.12.

Even though the results of this question were better this year than in previous years, an M/Mmax of 0.4 is still insufficient. The R_{it} value of 0.62 indicates that the question is distinctive [10]. The

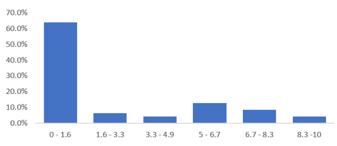
	max points	average	variance	M/Mmax	R_{it} exam
January 2019 (Question I)	10	1.59	8.64	0.16	0.56
January 2018	10	2.38	10.00	0.23	0.57
January 2020	10	4.2	18.37	0.42	0.62

Table 4.12: Result of question 2B.

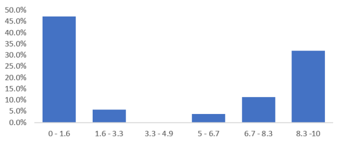
variance of this question has risen to 18, indicating that the spread to the average value is greater than in previous years. For a better understanding of the distribution of this question, the results are shown in three histograms in Figure 4.4. In clarifying the copy constructor, the debugger was



(a) Result of he copy constructor 2019.



(b) Result of he copy constructor 2018.



(c) Result of he copy constructor 2020.

Figure 4.4: The distribution of the results of the copy constructor questions in 2017, 2019 and 2020.

frequently used in both the lecture and the practical. This was partly successful, but it should have been better.

Conclusion Based on the improved results of the exam ”Introduction object-oriented programming”, the introduction of the debugger as an object-oriented learning tool in both the lecture and the practical part would be successful.

4.2 Analysis of the current ”Object-oriented programming part II” exam

The exam in the academy year 2020 was held differently due to unexpected circumstances. Due to the outbreak of the Coronavirus, it was not possible to hold the exam at the location of the Hague University. As such, the written exam was held online. Under normal circumstances (exam held on location without internet), students were allowed to keep books, sheets, dictations and a calculator. So they could look up C++ statements, but they could not use the compiler and discuss with each other.

To make fraud more difficult, the exam had been divided into four parts. The first part was the general information including the class diagram, which was visible all the time, the second, third and fourth part were set as an extra time lock. Besides, the variable names had to consist of the first two letters of the surname and the last two digits of the study number. As an example: Mr Been has a study number 123456.
int a; is wrong.
int be_a56; is right.
While we took measures to make the fraud as difficult as possible; the digital taking of the exam probably had an impact on the results.

The content of the ”Object-oriented programming part II”(OOPR2) course has not changed compared to the two previous years. The lectures have been adapted as described in Section 3.4.3, allowing the debugger to be used to clarify concepts of OOP. Unfortunately, the course coincided with the start of the Corona pandemic in the Netherlands. This had direct or indirect consequences, a number of these consequences were:

- Part of the lectures had to be taught online, so there was little or no interaction with the students.
- The lectures were followed less than during the course Introduction object-oriented programming. The online lectures were attended by no more than 20% of the students.
- When discussing the trial test online, 16 out of 50 students took part.
- The exam was taken online, as explained above.

The circumstances between this exam and that of a year ago are not quite the same, which makes the measurement less reliable.

To analyse the results of the Object-oriented part II exam, first, the final results of the exams and the two previous editions are compared with the results of the Introduction object-oriented programming exams. We did that to analyse whether the Object-oriented part II exam has a similar progression than the Introduction object-oriented programming exam. In the second part of this section, the results of the individual questions, which can be compared to the questions discussed in Section 3.3, are analysed.

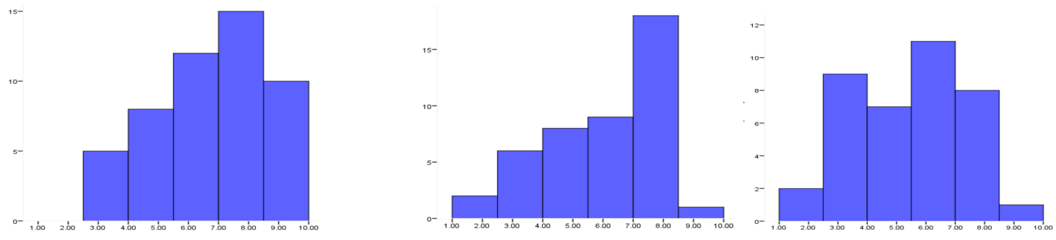
4.2.1 The results compared to previous years

Except for the problems caused by the Corona virus, there may be another difference with the same course of the last two years. Through the improved results of the course Introduction object-oriented programming, it can be assumed that at the beginning of this course, the knowledge in the domain of object-oriented programming may be better. Despite these limitations, it remains useful to analyse this exam. When we compare the results of the last OOPR2 exam with the two previous years, which are shown in Table 4.13, we see that the success rate of the last exam is 74.4%, which is 10.4% and 21% better than in the two previous years.

academic year		OOPR1	OOPR2
2019 - 2020	number of participants	53	50
	success rate	71.7%	74.0%
	average	6.5	6.7
	variance	4,7	2.8
2018 - 2019	number of participants	37	44
	success rate	51.4%	63.63%
	average	5.2	6.0
	variance	3.6	3.5
2017 - 2018	number of participants	38	38
	success rate	50.0%	52.6%
	average	5.1	5.3
	variance	3.7	3.5

Table 4.13: Results of both OOP courses.

When we place the distributions of the OOPR2 exam in a histogram, which can be seen in Figure 4.5, we see that none of the distributions is a normal distribution. We decided to use the



(a) Exam academic year 2019. (b) Exam academic year 2018. (c) Exam academic year 2017.

Figure 4.5: The distribution of the results of the exams object-orieted programming part II.

Mann-Whitney U test to determine p-value and effect size using the Hans Wendt formula [33, 56]. The results of these values are shown in Table 4.14. With the Mann-Whitney U test, a p-value of 0.085 and an effect size of 0.21 are obtained between the 2019 and 2018 exams, indicating no statistic significance between the two exams. In contrast, this is the case between the 2019 and 2017 exams with a p-value of 0.01 and an effect size of 0.41.

OOPR2		calculation by SPSS			Rank-Biserial Correlation r
comparing academic years		Mann-Whitney U	Z	Exact Sig. (2-tailed) (p-value)	
2019	2018	872.5	-1.725	0.085	0.21
2019	2017	595	-3.317	0.01	0.41

Table 4.14: Statistic results of between the exams OOPR2.

It is unclear why there is a effect size of 0.21 between the OOPR2 exams in 2019 and 2018 and a size effect of 0.41 between the 2019 and 2017 exams. A possible explanation could be that the students who did the exam in 2018 were better (motivated). The results of Table 4.1 show that this group of students also did better in related courses. From the results of Table 4.13, it does not appear that the Corona pandemic and the online exam had major consequences.

4.2.2 The results of the exam Object-oriented programming part II, 2020

The analysis of the exam Object-oriented programming part II shows that it is an acceptable exam with a Crombach alpha of 0.7 [10, 11]. The exam consists of eight questions and a total number of eighty points. An overview of the analysis of the exam up to question level is shown in Table 4.15. None of the questions had an M/Mmax lower than 0.5, and except for question two, all questions had the maximum score. Question two was an implementation of an interface in C++, the methods had to be declared pure virtual, and the virtual destructor had to be declared too. Furthermore, question two had an r_{it} of 0.0, which indicates that the question is not distinctive, but neither ambiguous. Questions four, five and eight will be analysed in more depth below and compared with similar questions from the 2019 exam, which discusses in Section 3.3. This exam was an open book exam and built around a class diagram which is shown in Figure 4.6. The students could choose their own way of implementation in C++. The complete exam can be found in Appendix E.

	total	1	2	3	4	5	6	7	8
maximum number of points	80	12	6	5	20	6	6	8	17
total number of points	4000	600	300	250	1000	300	300	400	850
total number of points by students	2543	383	190	216	703	235	168	248	400
avarage	50.9	7.7	3.8	4.3	14.1	4.7	3.4	5.0	8
variance	223.3	7.7	2.6	1.7	25.1	2.4	4.5	7.3	27.9
standard deviation	14.9	2.8	1.6	1.3	5.0	1.6	2.1	2.7	5.3
high score	72.0	12.0	5.0	5.0	20.0	6.0	6.0	8.0	17.0
M/Mmax	0.64	0.64	0.63	0.86	0.70	0.78	0.56	0.62	0.47
r_{it}		0.5	0.0	0.6	0.8	0.6	0.7	0.8	0.8
Cronbach’s alpha	0.7								

Table 4.15: Analyse of the results of exam part II April 2020.

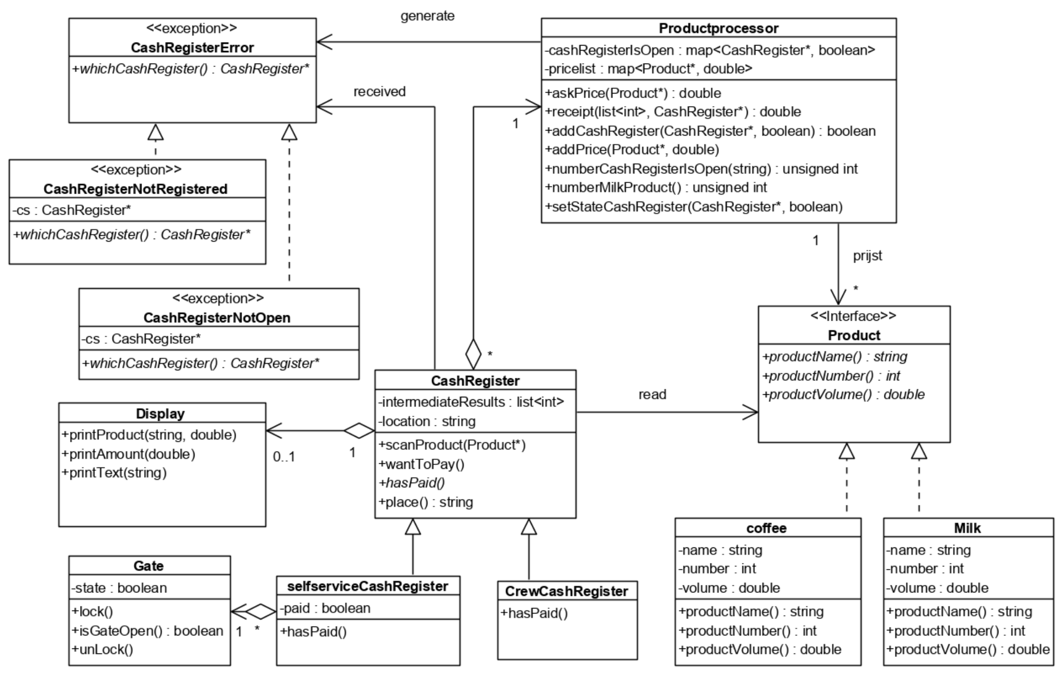


Figure 4.6: Class diagram of the exam Object-oriented programming part II.

With discussing the analysis per question, the question is repeated first. Then a possible solution is given. After this, the results per question are compared with the similar question a previous year, after which a further analysis takes place per item. An indication can be obtained by analysing per item to which parts of the object-oriented programming the debugger could have provided a clarified contribution.

Question four: Make the method: `void wantToPay();` of the class **CashRegister**.
One possible answer to question four is shown in Code fragment 4.1.

```
void CashRegister::wantToPay() {
 try {
  double finalAmount=vw->receipt(intermediateResults,this);
  if(displ) displ->printAmount(finalAmount); //check if a display is connected
        hasPaid();
 }
 catch(CashRegisterNotOpen& no) {
  if(displ) displ->printText("CashRegister␣is␣not␣open␣with␣location␣"+
                              no.whichCashRegister()->place());
 }
 catch(KassaNietGeregistreerd& ng) {
        if(displ) displ->printText("CashRegister␣is␣not␣registered␣with␣location␣"
                              + ng.whichCashRegister()->place());
 }
}
```

Code fragment 4.1: Answer to question four.

This question was the most extensive question with 20 out of 80 points and corresponds to question three of the April 2019 exam. The results of both questions are shown in Table 4.16. The

	max points	average	variance	M/Mmax	R_{it} exam
January 2019 (question 3)	20	10.5	33.6	0.52	0.75
January 2020 (question 4)	20	14.4	24.9	0.70	0.80

Table 4.16: Result of question four.

students did perform better on the question in 2020, the M/Mmax increased from 0.52 to 0.70, the variance decreased from 33.6 to 24.9, and the r_{it} increased from 0.75 to 0.80. Although the average score on this question was considerably higher than the similar question of the exam in 2019, this question kept its distinctiveness.

The question consisted of three main parts:

- Calling of a method with `this` pointer as a parameter and the calling of its abstract function.
- Using `try` and `catch` to capture the exception.
- Display information by calling a method of the class **Display**.

The M/Mmax and the r_{it} of each of these parts are shown in Table 4.17. When we look at the M/Mmax, we see that the exam was done better during the research than the year before, while each component retained its distinctive character. Despite the difficulties in lecturing and taking the exam

calendar year	2019		2020	
	M/Mmax	r_{it}	M/Mmax	r_{it}
Calling method, `this` pointer and calling abstract method	06	0.9	0.8	0.8
`Try` and `catch`	0.7	0.8	0.8	0.8
Calling a method of another class (Display)	0.3	0.6	0.5	0.8

Table 4.17: The mean/Max per parts of question four.

online, due to the Corona pandemic, the trend of improved results that started in the previous course remained.

Question five: Make the method: `bool addCashRegister(CashRegister*, bool);` of the class **Productprocessor**.
One possible answer to question five is shown in Code fragment 4.2.

```
bool Productprocessor::addCashregister(CashRegister* cashr, bool statuscr) {

        std::pair<map<CashRegister*,bool>::iterator,bool> ret;
        ret=cashRegisterIsOpen.insert(pair<CashRegister*,bool>(cashr,statuscr));

        return ret.second;
}
```

Code fragment 4.2: Answer to question five.

This question was about the insertion of data in a map, where it had to be checked if the insertion was successful. Table 4.18 shows that the question was done better than the similar question last year, where the M/Mmax increased from 0.42 to 0.78. However, the question has become less distinctive,

	max points	average	variance	M/Mmax	R_{it} exam
January 2019 (question 5)	6	2.5	2.4	0.42	0.71
January 2020 (question 5)	6	4.7	2.4	0.78	0.62

Table 4.18: Result of question five.

the R_{it} decreased from 0.71 to 0.62. But there is still a positive correlation in the distinctive character [10].

As mentioned in the previous paragraph, the question consists of two main parts:

- To insert the data under a key in the map.
- To check whether the insertion was successful.

calendar year	2019		2020	
	M/Mmax	r_{it}	M/Mmax	r_{it}
Insertion in the map	0.6	0.8	0.8	0.8
check insertion is successful	0.2	0.6	0.9	0.8

Table 4.19: The result per main item of question five.

When we compare the results of the two main components, which are shown in Table 4.19, we notice that the return value (the check, ”whether the insertion in the map was successful”), is increased from 0.2 to 0.9. The r_{it} of both parts are 0.8, which is all right. To show why it needed to check, ”if the insertion was successful”, with the help of the debugger, could maybe a positive effect on the result.

Question eight: Make the method: `double receipt(list<int> productnumbers, CashRegister* );` of the class **Productprocessor**.
One possible answer to question eight is shown in Code fragment 4.3.

```
double Productprocessor::receipt(list<int> l,CashRegister* k) {
  double total=0;
  auto ks=cashRegisterIsOpen.find(k);
  if (ks==cashRegisterIsOpen.end())
        throw CashRegisterNotRegistered (k);
  if(!ks->second)
```

```
        throw CashRegisterNotOpen(k);

    for(auto i:l) {
      int temp=i;
      for(auto p:prijslijst) { //search to product number
        if((p.first)->productNumber()==temp) //is this the right product number
          total += p.second;    //add the price to the total
          }
    }
    return total;
  }
```

Code fragment 4.3: Answer to question eight.

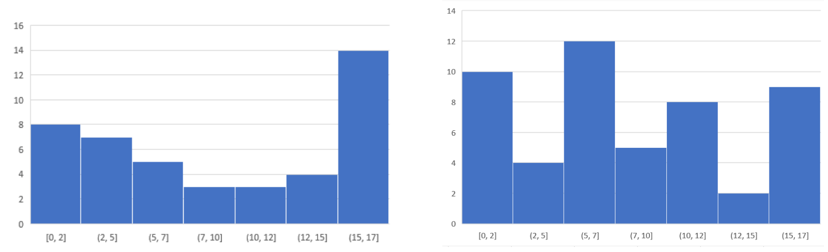
This question was extensive and was about working with a map and a list and creating an exception. Question eight is similar to question six of last year's exam. The results of both questions are shown in Table 4.20. The result of this question was no better than the one of the previous year. Both

	max points	average	variance	M/Mmax	R_{it} exam
January 2019 (question 6)	17	8.9	36.9	0.52	0.87
January 2020 (question 8)	17	8	27.9	0.47	0.79

Table 4.20: Result of question eight.

the M/Mmax and the r_{it} are lower than the year before. The variance is also lower, which indicates that there is less spread.

To clarify the difference in distribution, the results of both questions are shown by the histogramin Figure 4.7. Despite the histogram difference, the p-value after the Mann-Withney U-test is 0.48 and U is 1006, which results in an effect size of 0.08. Which indicates that both distributions do not differ significantly. The histogram of Figure 4.7a looks like a bimodal distribution [1] with the left peak of students who did not understand the material and the right peak of students who understood the material. If we look to histogram 4.7b, it seems that the pattern of this question can be taken out. Because this question consists of three parts, the distribution of the histogram could maybe indicate that the student did not understand anything or understood one, two or three parts.



(a) Distribution of question six part2 2019. (b) Distribution of question eight part2 2020.

Figure 4.7: Difference in distribution between question six in 2019 en question eight in 2020.

As noted in the previous paragraph, this task consists of three parts, which are:

- search to key in a map.
- throw exceptions.
- using STL iterators.

calendar year	2019		2020	
	M/Mmax	r_{it}	M/Mmax	r_{it}
Search to key in a map	0.62	0.74	0.64	0.82
Throw exceptions	0.70	0.85	0.61	0.78
Using STL iterators	0.45	0.93	0.42	0.93

Table 4.21: The mean/Max per parts of question eight year 2020.

The M/Mmax and the r_{it} of each part are shown in Table 4.21. The statement ”throw an exception” as is shown in Code fragment 4.3 did in 2020 worse than in 2019, while the other two parts have about the same result. While capturing of the exception (try and catch) in question four did better in 2020, as can be seen in Table 4.17. Unfortunately, the results of this question have not improved. The extent to which the students have come to terms with the debugger is unknown. The lectures were digitally recorded and could be viewed.

Conclusion Although the average result of the written exam increased from 6.03 to 6.73, it is not possible to get an indication of the extent to which the use of the debugger to clarify object-oriented programming in this exam has led to a better result. During the course period, the Corona pandemic became more intense, so several lectures were taught online, and the exam was held online. These unexpected complications make the measurement whether the debugger has contributed to a better understanding of object-oriented programming less reliable. It is also possible that the improved results of the previous exam (the course Introduction object-oriented programming), contributed to the improved results of this exam.

4.3 Evaluation of using the debugger during the course by the students

The evaluation by the students about using the debugger in the OOPR1 course was done by means of a questionnaire based on a Likert scale of five levels [9, 34] with the items, strongly agree, agree, neutral, disagree, strongly disagree. We also included a few open questions. During the last lecture, the students completed the questionnaire. Unfortunately, there were only sixteen students present. Nevertheless, the feedback was useful to get information about what was good and what could be improved.

The questionnaire consisted of three parts. One for the evaluation of the multiple-choice questions at the beginning of the lecture, which consisted of five multiple-choice questions and two open questions. One for the lecture itself, which consisted of nine multiple-choice questions and three open questions and one for the lab sessions, which consisted of eleven multiple-choice questions and two open questions. All three parts gave a positive view.

4.3.1 Evaluation of the multiple-choice questions at the beginning of the lectures.

The students generally evaluated the multiple-choice-questions at the beginning of the lecture as positive, as is shown in Figure 4.8. The questions related to a review of the previous college. The multiple-choice question were:

1. Were the multiple-choice questions at the beginning of the lecture an added service as a look back on the treated material of the previous lesson.
2. The questions in which the debugger played a role in the form of *"which line does the debugger jump to when push 'step into' button"* were clear.
3. The questions in which the debugger played a role in the form of *"which line does the debugger jump to when push 'step into' button"* resulted in a better insight into the invocation of respective constructor, destructor or methods.
4. The questions where the content (the attributes) of an object was visually displayed using the DDD debugger were clear.
5. The questions where the content (the attributes) of an object was visually displayed using the DDD debugger resulted in a clearer picture of the content of the object.

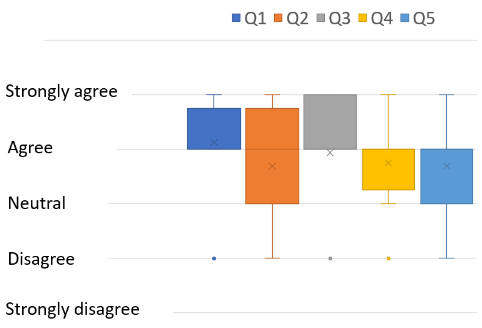


Figure 4.8: Results of the m.c. questions about using the debugger with the review of previous lecture.

The students were satisfied with the look back with multiple-choice questions. Especially the questions where you see what would be the next statement. The two open questions where:

6. Tips regarding the multiple choice questions.
7. Flops regarding the multiple choice questions.

Eight out of sixteen students answered at least one of the two open questions. One student wrote that it does not matter. Four students wrote that some questions could be more transparent, two students wanted to be able to see the questions after the lecture as well, three students found it instructive and one student wanted more time to fill in the questions.

4.3.2 Evaluation, using the debugger during the lectures.

To determine what the students thought about using the debugger during the lecture. We included seven multiple-choice questions with a five-level Likert-scale and one multiple-choice question with the items: did not do anything, just downloaded, unpacked and watched, compiled and run, and worked with the debugger in the questionnaire. Furthermore, we also included an open question about the topics using the debugger by themselves, and there were two open questions with tips and flops about using the debugger as a demonstration tool during the lecture in the questionnaire.

The results of the seven multiple-choice questions are shown in Figure 4.9. The multiple-choice questions were:

8. Switching between the sheets and the debugger tool environment during the presentation is disruptive.
9. Visually displaying the data and pointers that refer to other objects is enlightening.
10. Visually showing that a base class is a part of the derived class is enlightening.

11. By using a visual debugger (DDD) during the lecture, a better insight is gained into the complexity of the substance.
12. By walking through the program step by step with the debugger tool, it has become more evident that the constructor of the base class is also called.
13. By walking through the programme step by step with the debugger tool, the polymorphism (dynamic binding of methods) has become more transparent.
14. The use of a debugger during the lecture is an addition.

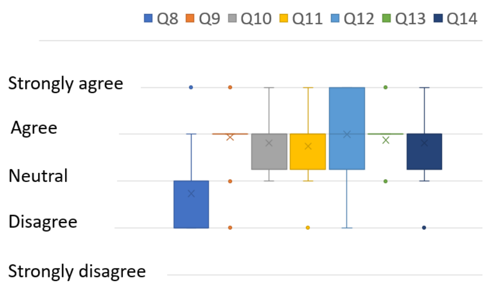


Figure 4.9: Results of the m.c. questions about using the debugger during the lecture.

The results of figure 4.9 indicate that students were generally positive about using the debugger during lectures. Only two students of the sixteen found the changes between the presentation and the debugging environment disruptive or intensely disruptive. While things did not always go fluently and problems occurred with the presentation of the debugging environment on the beamer. One student each indicated to be dissatisfied with questions 9, 11 and 13 and two students with question 12. The questions relate to the DDD debugger (question 9, 10 and 11) were mainly assessed between neutral and agree, which corresponds to the results for the same type of questions about the previous lecture (questions 4 and 5). The students appreciated questions 12 en 13 in which the tracer had a role, higher. Although multiple-choice questions nine to fourteen show that students were satisfied with using the debugger during the lecture, they were less satisfied than those from multiple-choice questions 1 to 5. One possible explanation could be that the changes between presentation and debugging environment were less smooth than indicated.

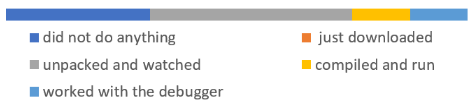


Figure 4.10: Using the example code by themselves.

The students were also asked whether they used the sample code that had been discussed during the lecture. Two-thirds of the students downloaded the example code, part of them did some work, and a small part worked with the debugger, as can be seen in Figure 4.10. There was also an open question about whether they use the debugger by themselves. The reaction was diverse: about twenty percent was negative, thirty-five percent neutral, thirty-five percent positive, two students did not answer question. Thirteen students answer one of the two questions about tips and flops. There were a few recommendations such as unclear text and switching between presentation and debugger, there was a contradiction, one student would focus more at the debugger and another found screenshots enough. Despite the debugger was successful during the lecture, it should be improved so that the students keep them attention.

4.3.3 Evaluation, using the debugger during the lab sessions.

The use of the debugger during the lab was experienced as positive by the students. Contrary to previous years, where the debugger was only compulsory for the first assignment, the students were obliged to use the debugger in four out of five assignments. Another difference was that in the past the debugger was only used to search for errors in a program, in this research the debugger was mainly used to get a better insight in object-oriented programming by visualizing objects and using the tracer. The last part of the questionnaire focused on the lab. There was one specific question about walking through a programme step by step (assignment 1), and whether the student already had experience with using the debugger. There were eight questions about the visual representation of objects and their relationships, two questions about searching for errors and two open questions with tips and flops.

The first question was about the experience and using a debugger in general, like walking through a program step by step. Figures 4.11 shows the result of this question, which was: The first assignment of the lab was about using the gdb debugger, including turning on a LED step by step. (multiple answers are possible for this question)

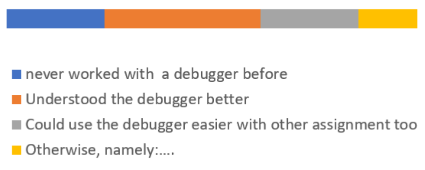


Figure 4.11: Results of question about using debugger(question 1).

1. I had never worked with a debugger before.
2. Because of this assignment, I understood some of the debugger's possibilities.
3. This assignment made it easier for me to work with the debugger for the other assignments.
4. Otherwise, namely: ....

About fifty per cent of the students found that they understood the debugger better, and thirty per cent found that they could easier use the debugger after assignment one was done. Four out five students who declared that they had never worked with a debugger before, indicated that they also understood the debugger better after doing assignment one. About twenty per cent said that they had used another debugger earlier but not which. Because of the students' satisfaction with assignment one, it is not advisable to skip this assignment and start immediately with the visualization of objects and their relationships.

The main part of the questions about the lab consisted of questions about using the visual debugger DDD and the screenshots that the students had to take. It were eight five-level Likert-scale questions, which were:

2 By including screenshots of the debugging environment in the assignment description, the purpose of the assignment came across more clearly.

3 By walking through the program with the debugger, the concepts of class, objects and methods have become more evident.

4 By visualizing one or more objects with the DDD debugger (assignment two, class Duration is a composition by class Logled), the concepts of composition and aggregation have become more evident.

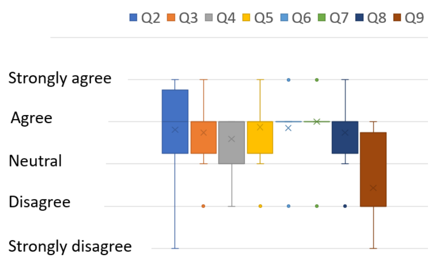


Figure 4.12: Results of the m.c. questions during the lab sessions.

5 By visualizing one or more objects with the DDD debugger (assignment two, class Duration is a composition of class Logled), it has become more evident when a pointer should or should not be used during programming.

6 The concept of inheritance, as in assignment three (a SingleLED is an LED) has been clarified by visualising the object with the DDD debugger.

7 By visualizing objects, it has become more evident why the copy constructor needs to be overloaded in some cases.

8 Showing the content of an object through screenshots had a clarifying effect.

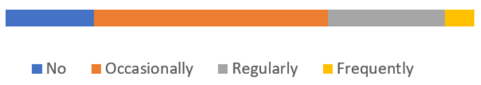
9 Making the screenshots was unnecessary; only the program code was clear enough.

Figure 4.12 shows the results of these five-level Likert-scale questions. Despite making screenshots was more work for the students, the students were quite positive and found it useful, as the result of question 9 shows. The debugger clarified general OO terms such as class, objects and methods (question 3) and by visualizing objects, more profound concepts such as inheritance and difference between composition and aggregation have become clearer too(questions 4 and 6). More OO programming language (C++) dependent concepts such as the use of pointers(question 5) and copy constructor overloading(question 7) have also become clearer through visualization. The screenshots were also a clarification in the assignment description and in showing the content of an object. In addition to these eight questions related to the screenshots, there were two multiple-choice questions that specifically addressed searching for errors in a program.

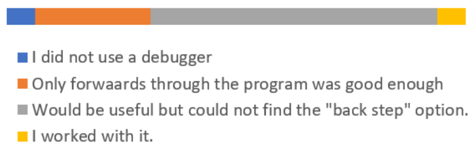
To find out whether students used the debugger to look for errors in a program. The questionnaire had two questions about this topic which are:

10 Did you not only use the debugger for screenshots but also to look up errors in your program? (this could be another debugger besides the DDD debugger).

11 If you used the debugger to look for errors in your program, would a "back step" option be useful?



(a) Debugger used for errors too(question 10).



(b) Using back step.

Figure 4.13: Debugger used with searching for errors in program(questions 10 and 11).

From the results of both questions, which are shown in Figure 4.9, it appears that most of the students, who completed the questionnaire, used the debugger. Because the assignments are mainly focused on the principles of object-oriented programming and less on the application of an algorithm, it could be a reason that the debugger did not often use by the students. Although the "back step" was unknown to the students, they indicated that it would be useful. That many students used the debugger at least a few times, could possibly have a positive effect on the debugging of more complex algorithms.

The open questions were about the tops and flops of the lab sessions. The answers to the open questions were ambiguous and contradictory on several subjects. One student likes the fact that the practical tools were also discussed in the lecture. Another student writes "making clear what the intention is and linking it to the lecture". Two students were positive about taking screenshots. Two other students thought the screenshots were unnecessary because they already understood the subject or because they drew it wide themselves. A quarter of the students indicated to have had problems with the working environment (working remotely on a Raspberry Pi). The answers to the two open questions partly gave the same view as the multiple-choice questions but also offered the possibility for additional information, such as the fact that the screenshots are a limitation in making your own solutions. The fact that improvements to the working environment were needed was a confirmation that could be heard during the practical lessons.

During the practical course, at least twenty per cent of the students had problems with the practical-environment. Two students thought the DDD debugger was old-fashioned and one made the following noticed that the last release of the DDD debugger was when they were still in primary school.

4.3.4 Conclusion

Most of the students who completed the list were satisfied with using the debugger during the course OOPR1. The use of the debugger during all three parts (looking back on the previous lecture by using multiple-choice questions, during the lecture and the practical) was experienced as positive. Especially the graphical representation of OO concepts with the DDD debugger by using screenshots. Although there were some minus points during the practical lessons such as: why take screenshots if the object-oriented part was clear. Besides, the working environment with remote Raspberry Pi and Eclipse as a development environment caused problems for several students. Because a majority of the students were satisfied with the debugger, it is recommended to continue with it and make the practice environment more attractive.

Chapter 5

Discussion

The discussion section aims to analyze the results of Chapter 4 on a more abstract level. First, we discuss the results of the changing in a broad perspective. Second, the results of changing the course ”Introduction object-oriented programming”. Third the results of changing the course ”Object-oriented programming part II”, forth further research and finally the treads and validation.

5.1 Interpretation of results

This section provides an overview of the results obtained by the students in the System Development line of the study programme Network System Engineering, which runs from the second semester of the foundation year up to the end of the second academic year. The order of the programmes is shown in Figure 5.1, which does not include the first semester of the foundational year. This is a common orientation semester of five computer science programmes. Furthermore, the third and fourth years are not included. This is because these are an internship, a major, a minor and graduation.

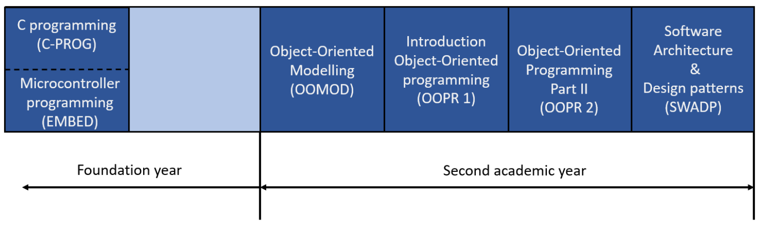

C programming (C-PROG)
Microcontroller programming (EMBED)
Object-Oriented Modelling (OOMOD)
Introduction Object-Oriented programming (OOPR 1)
Object-Oriented Programming Part II (OOPR 2)
Software Architecture & Design patterns (SWADP)
Foundation year
Second academic year

Figure 5.1: The courses of the system development line.

The results of the average score and success rates of the various courses in the system development line of the last three years are shown in Table 5.1. The content of each course did not change the last three years, and except for the OOMOD course, all courses did not change lecturers in all three years. For the C-PROG[1] and EMBED[1] courses, no account has been taken of students who left the study programme at the end of the academic year, which had a negative impact on the average and success rate.

Furthermore, the OOPR2 and SWADP(Software Architecture & Design Patterns) courses of the 2018 cohort suffered from the Corona pandemic in which the OOPR2 exam was taken online.

[1]form previous academic year.

academic years		C-PROG[1]	EMBED[1]	OOMOD	OOPR1	OOPR2	SWADP	average of the courses
2019 - 2020	success rate	64%	48%	59%	72%	74%	60%	62.8%
	average	5.6	4.9	5.3	6.5	6.7	5.8	5.8
2018 - 2019	success rate	64%	63%	71%	51%	63%	70%	63.7%
	average	6.4	5.8	6.2	5.2	6.0	5.9	5.9
2017 - 2018	success rate	41%	52%	63%	50%	53%	40%	49.8%
	average	4.7	4.9	5.8	5.1	5.3	5.1	5.2

Table 5.1: Results of the exams in the field of object-oriented programming cohort 2018.

If we only look at the average scores of the courses from Table 5.1, which is shown in Figure 5.2, we notice that both object-oriented programming courses of the academic years 2019 did better than the other courses. Where in the other two years the score of the object-oriented programming

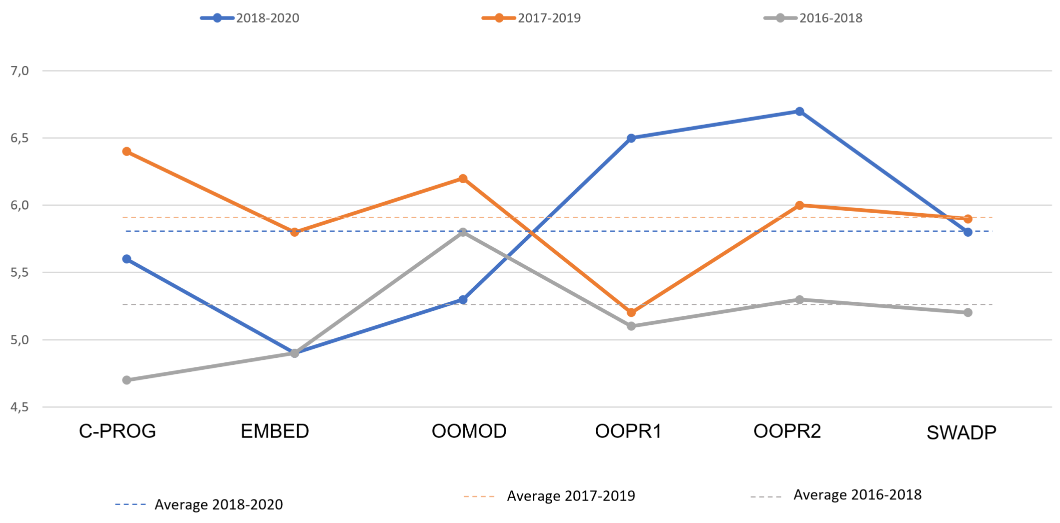


Figure 5.2: The average score of the courses in the same field as object-oriented programming.

courses were below or around the overall average (dashed line), the score in the academic years 2018-2020 is higher. On the basis of the results from Table 5.1, the hypothesis **People who use a debugger when learning object-oriented programming better understand the concepts** can be accepted, although repeating the experiment is desirable.

Although this research shows that the results in the field of object-oriented programming have improved positively, after the debugger has been used as a learning tool, in particular, to visualize objects and to walk step by step through example program code. This does not mean that the students have a better understanding of object-oriented design. Figure 5.2 shows that the results of the follow-up course Software Architecture and Design Patterns (SWADP) have not improved. However, the lectures were given online by the Corona pandemic rather than in the classroom.

The difficulties with object-oriented programming. In order to find out what the most common problems with object-oriented programming were, the last examinations of both courses have been analysed (**RQ3**). Most of the resulting object-oriented problems were:

- The principle of inheritance, such as the keyword virtual and calling the basic constructor during the invocation of the derived constructor.
- Invoking methods of a different class.
- Initialisation of attributes.
- The copy constructor.
- To apply the STL features such as using an iterator and calling the right method.

To clarify these problems by using a debugger (**RQ4**), we used two different debuggers; the DDD debugger to visualize the objects and the Eclipse IDE with the GDB debugger for going through a programme step by step in combination with watching attributes and remote debugging. Furthermore, the Eclipse IDE is open source and works on multiple platforms. How these problems could be clarified with help of the debugger, is described in Section 2.2 ”The debugger as a clarification tool for object-oriented concepts” and in Sections 3.2 and 3.3 ”Analysis of the exams 2019”.

This research has been limited to the object-oriented components taught at the NSE department of the academy I T & Design of The Hague University of Applied Sciences. Further research will have to show whether using the debugger as a learning tool can be used in other concepts of object-oriented programming.

The problems with using the debugger. Because many professional programmers and students find working with a debugger difficult and prefer to use traditional ”printf” debugging [7], a way had to be found so that students could not avoid the debugger during the lab sessions(**RQ2**). This was done by having students take screenshots of visualised objects with their study number and the Unix/Linux Epoch time as attributes. These screenshots had to be uploaded to the learning environment Blackboard. An extra check to prevent that students could not take each other’s screenshot during the practical course was the automatic creation of the value of an attribute, where the numbers of the study number are added up. In post-check tests, all Epoch times turned out to be different, and the product numbers were equal to the addition of the digits of the study number.

The fact that the Epoch times were different indicates that they were made at different times, and the students probably did not take each other’s screenshots. This does not mean that the programme code or parts of it were copied from each other. A disadvantage of checking the screenshots afterwards is that this is time intensive.

Working with the debugger. To ensure that the students had sufficient skills to use a debugger(**RQ1**) so that they could concentrate better on the object-oriented concepts, we decided to start with the screenshots at the second assignment of the lab sessions. We did not change the first assignment, which was already working with the debugger. Furthermore, during several lectures, we paid extra attention to the DDD debugger.

During the lectures, we used the DDD debugger to visualize the objects, and the Eclipse debugger to walk through a program code step by step to clarify the object-oriented code. To keep the attention of the students, switching between presentation and debugger environment had to be fluent. The questionnaire as discussed in Section 4.3.2, showed that the students were not dissatisfied with switching between the debugging environment and the presentation.

Further research should be done into the effect of visualization earlier in the curriculum, as in C programming. Matthew Heinsen had positive results in his recent paper [28] and in his PhD [21]. They developed a visualisation and debugging tool for novice C programming, named SeeC. This tool combines visualisation and debugging technique in one system. Some results of student satisfaction with regard to virtualization can be seen in figure 5.3. The tool is still under development (latest version 0.40[2]) at the moment.

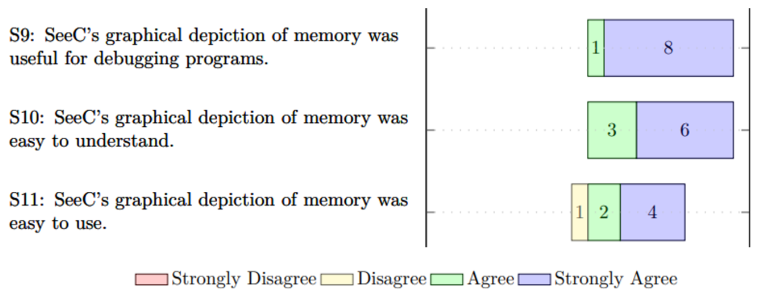


Figure 5.3: Some results from the PhD of Matthew Heinsen [21].

5.2 Threats to validity

In this section, we discuss both the limitations and the validity of our research. Furthermore, by identifying possible points for improvement, areas for future research or possible weaknesses can be improved.

The limitation of this study is that only the data from the written examinations are included in the analysis. This gave insight into whether the theory was better understood than in previous editions, but the practical skills of programming remained underexposed as a result. This disadvantage was reduced because the exams were structured in such a way that programming skills could be applied.

Construct validity of this research is the number of students who participated in this research. In the exam Introduction object-oriented programming, there were fifty-three, and in the exam Object-oriented programming part II, fifty. Due to this small number of students and the diversity in previous education within the group, we decided not to split the group into two equal groups, but to treat it as a single group and compare it with the exams of a previous year. In order to be able to measure as accurately as possible, we have included the results of two previously taught courses from the same discipline in the analysis.

In order to get feedback from the students, we had drawn up a questionnaire and asked the students to complete the questionnaire during the last lecture that is normally well attended, because the trial exam Introduction object-oriented programming was discussed. Unfortunately, only 30% of the students were present, one of the reasons was the competition between the delivery of the project and the discussion of the trial exam. It is advisable to offer the questionnaire to both the second last lecture and the last one so that those who have not yet completed the questionnaire can still complete it. Furthermore, no account has been taken of the fact that the group process can play an essential role in the results obtained [24, 13].

Internal validity refers to the threats inherent in the research described. A threat was that the students would not use the debugger during the practical course, or used it sporadically. As a result, the impact of the debugger on the understanding of object-oriented programming would not be

[2]https://seec-team.github.io/seec/releases.html

sufficiently expressed. By having the students take screenshots of situations that were only possible with the debugger in combination with their name and study number as an attribute, it was difficult for the students to avoid the debugger. Another threat was the complexity of remote programming and debugging. To reduce the complexity, the students were offered a virtual environment in which the necessary software and settings were done. Despite these precautions, problems still occurred. It is advisable to simplify the network environment by, for example, using the laptop as a hotspot and allowing the Raspberry Pi to make contact with the laptop.

External Threats refer to the extent to which the results of this study can be generalized. The exam of the course object-oriented programming part II was kept online by the Corona pandemic. To minimize fraud, we split the exam into 3 parts, each part having a time lock. Besides, the name of each variable had to contain the first two letters of the surname and the last two digits of the study number. Because the course contains specific C++ features, it will need to be adapted for JAVA and C# programming environment.

Chapter 6

Conclusion and Future Work

The aim of this research was whether the debugger tool could be used to clarify the concepts of object-oriented programming for novice students. A better result was achieved on both courses than in the two previous years, whereas this was not the case for previous courses that related to both object-oriented programming courses with the same students. This approach could be useful to extend our understanding of teaching object-oriented programming to novice students.

The course Introduction object-oriented programming, which was the first part of our study, showed that the debugger could not only be used to find errors in a programme but could also probably be used to clarify the abstraction of object-oriented programming in C++. We introduced the debugger during the lecture and expanded the lab sessions so that the students had to use the debugger in four out of five assignments. During the lectures, the debugger was used to explain new topics and to look back at the treated topics from the previous lecture. During the lab sessions, the students had to use the debugger mainly to make particular object-oriented situations visible.

The average grade increased from 5.1 and 5.2 in the two previous years to 6.45, while the scores of the same group of students on related courses had not progressed. To get the students' opinion, the students had to fill in a questionnaire in the last lecture of the course Introduction object-oriented programming. Although seventeen students filled in the questionnaire, the findings suggest that the students were satisfied with using the debugger to clarify concepts of object-oriented programming. The intensive use of the debugger as a helpful tool in understanding the basic abstraction of object-oriented programming was constructive.

The course object-oriented programming part II, which was the second part of our research, also shows a better result than the two previous years, although the improvements are less than in the course ”Introduction object-oriented programming”. The average grade increased from 5.3 and 6.0 in the two previous years to 6.7.

In this course, the number of lecture hours was increased from one to two hours per week, which allowed us to allow students to work with the debugger during the lecture. Unfortunately, this part was strongly influenced by the Corona pandemic, which resulted in the last three lectures being offered online. Because the lectures were offered online, less than twenty percent of students attended the course. Answering individual questions to the students, if applicable, was done with the help of the debugger. The fact that the results were better with this course, even though the influence of the debugger is less, could be partly due to the results of the course Introduction object-oriented programming, which resulted in a higher entry-level of the students.

Future work

Further experimental research is needed to make a reasonable estimate of the extent to which the use of the debugger by the students during the lecture can contribute to clarifying more complex concepts of object-oriented programming.

There are a number of important changes which need to be made. For example, the practical environment should become simpler and more stable, and the check whether the student has applied the debugger correctly should no longer be done by means of screenshots, or the recognition of the correct screenshots will have to be automated.

Bibliography

[1] Bimodal Distribution: What is it? - Statistics How To. URL `https://www.statisticshowto.com/what-is-a-bimodal-distribution/`.

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, pages 84–88, 2005.

[3] Arwa A Allinjawi, Hana A Al-Nuaim, and Paul Krause. Evaluating the effectiveness of a 3d visualization environment while learning object oriented programming. Journal of Information Technology and Application in Education, 3(2):47–56, 2014.

[4] Oivind Andersson. Experiment!: planning, implementing and interpreting. John Wiley & Sons, 2012.

[5] Jim Arlow and Ila Neustadt. UML 2 and the unified process practical object-oriented analysis and design. Pearson Education, 2005. ISBN 9780321321275.

[6] LB Becker, M Gergeleit, and E Nett. Approach for implementing object-oriented real-time models on top of embedded targets. In OMER-2–Workshop on OO Modeling of Embedded RT Systems, 2001.

[7] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the dichotomy of debugging behavior among programmers. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pages 572–583, May 2018.

[8] Jens Bennedsen and Carsten Schulte. Bluej visual debugger for learning the execution of object-oriented programs? Trans. Comput. Educ., 10(2):8:1–8:22, June 2010. ISSN 1946-6226. doi: 10.1145/1789934.1789938. URL `http://doi.acm.org/10.1145/1789934.1789938`.

[9] Dane Bertram. Likert scales. Retrieved November, 2:2013, 2007.

[10] Lia Bijkerk. Basis Kwalificatie Examinering in het hoger beroepsonderwijs. Springer, 2015.

[11] J Martin Bland and Douglas G Altman. Statistics notes: Cronbach's alpha. Bmj, 314(7080): 572, 1997.

[12] Kim B Bruce. Controversy on how to teach cs 1 a discussion on the sigcse-members mailing list. In Working group reports from ITiCSE on Innovation and technology in computer science education, pages 29–34. 2004.

[13] LILIAN YA-HUI CHANG. Group processes and efl learners' motivation: A study of group dynamics in efl classrooms. Tesol Quarterly, 44(1):129–154, 2010.

[14] Chiu-Liang Chen, Shun-Yin Cheng, and Janet Mei-Chuen Lin. A study of misconceptions and missing conceptions of novice java programmers. In Proceedings of the International Conference on Frontiers in Education Computer Science and Computer Engineering (FECS), page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[15] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. Computer Languages, Systems & Structures, 44:89 – 113, 2015. ISSN 1477-8424. doi: https://doi.org/10.1016/j.cl.2015.08.005. URL `http://www.sciencedirect.com/science/article/pii/S1477842415000561`. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[16] Barry Cornelius et al. Java versus c++, 2004.

[17] J. H. I. I. Cross, T. D. Hendrix, and L. A. Barowski. Using the debugger as an integral part of teaching cs1. In 32nd Annual Frontiers in Education, volume 2, pages F1G–F1G, Nov 2002. doi: 10.1109/FIE.2002.1158137.

[18] DNM De Gruijter. Toetsing en toetsanalyse. Leiden: ICLON, Sectie Onderwijsontwikkeling Universiteit Leiden, 2008.

[19] Agostinho de Medeiros Brito Jr and Adelardo Adelino Dantas de Medeiros. A motivating approach to introduce object-oriented programming to.

[20] Thomas Dupriez, Guillermo Polito, and Stéphane Ducasse. Analysis and exploration for new generation debuggers. In Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies, IWST '17, pages 5:1–5:6, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5554-4. doi: 10.1145/3139903.3139910. URL `http://doi.acm.org/10.1145/3139903.3139910`.

[21] Matthew Heinsen Egan, Christopher McDonald, and Amitava Datta. Advanced debugging and program visualization for novice c programmers. 2015.

[22] O. Ezenwoye. What language? - the choice of an introductory programming language. In 2018 IEEE Frontiers in Education Conference (FIE), pages 1–8, Oct 2018. doi: 10.1109/FIE.2018.8658592.

[23] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. Computer Science Education, 18(2):93–116, 2008. doi: 10.1080/08993400802114508. URL `https://doi.org/10.1080/08993400802114508`.

[24] Donelson R Forsyth. Group dynamics. Cengage Learning, 2018.

[25] Stavroula Georgantaki and Symeon Retalis. Using educational tools for teaching object oriented design and programming. Journal of Information Technology Impact, 7(2):111–130, 2007.

[26] R. V. Golhar, P. A. Vyawahare, P. H. Borghare, and A. Manusmare. Design and implementation of android base mobile app for an institute. In 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pages 3660–3663, March 2016. doi: 10.1109/ICEEOT.2016.7755391.

[27] Said Hadjerrouit. A constructivist approach to object-oriented design and programming. In Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE '99, pages 171–174, New York, NY, USA, 1999. ACM. ISBN 1-58113-087-2. doi: 10.1145/305786.305910. URL `http://doi.acm.org/10.1145/305786.305910`.

[28] Matthew Heinsen Egan and Chris McDonald. An evaluation of seec: a tool designed to assist novice c programmers with program understanding and debugging. Computer Science Education, pages 1–34, 2020.

[29] Peter Hubwieser and Marc Berges. Minimally invasive programming courses: Learning oop with(out) instruction. In Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education, SIGCSE '11, pages 87–92, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0500-6. doi: 10.1145/1953163.1953195. URL `http://doi.acm.org/10.1145/1953163.1953195`.

[30] Achi Ifeanyi Isaiah, Agwu Chukwuemeka Odi, Alo Uzoma Rita, Anikwe Chioma Verginia, and Okemiri Henry Anaya. Technological advancement in object oriented programming paradigm for software development. International Journal of Applied Engineering Research, 14(8):1835–1841, 2019.

[31] Nicolai M Josuttis. The C++ standard library: a tutorial and reference. Addison-Wesley, 2012.

[32] Ayaan M. Kazerouni, Riffat Sabbir Mansur, Stephen H. Edwards, and Clifford A. Shaffer. Student debugging practices and their relationships with project outcomes. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19, pages 1263–1263, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3293794. URL `http://doi.acm.org/10.1145/3287324.3293794`.

[33] Dave S Kerby. The simple difference formula: An approach to teaching nonparametric correlation. Comprehensive Psychology, 3:11–IT, 2014.

[34] Roman Lewandowski. Evaluating the interdependent effect for likert scale items. In Business Information Systems Workshops: BIS 2019 International Workshops, Seville, Spain, June 26–28, 2019, Revised Papers, volume 373, page 26. Springer Nature, 2020.

[35] Xiangqi Li and Matthew Flatt. Medic: Metaprogramming and trace-oriented debugging. In Proceedings of the Workshop on Future Programming, FPW 2015, pages 7–14, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3905-6. doi: 10.1145/2846656.2846658. URL `http://doi.acm.org/10.1145/2846656.2846658`.

[36] D. Bennett M. Al-Jepoori. Understanding of the programming techniques by using a complex case study to teach advanced object-oriented programming. ICCE 2018, 2018.

[37] David J Malan and Henry H Leitner. Scratch for budding computer scientists. ACM Sigcse Bulletin, 39(1):223–227, 2007.

[38] A. Mallia and F. Zoffoli. C++ Fundamentals: Hit the ground running with C++, the language that supports tech giants globally. Packt Publishing, 2019. ISBN 9781789803907.

[39] Bruce Martin. The separation of interface and implementation in C++. Hewlett-Packard Laboratories, 1990.

[40] Zivorad M Milenovic. Application of mann-whitney u test in research of professional training of primary school teachers. Metodicki obzori, 6(1):73–79, 2011.

[41] Ms Nahlah, Mr Paramudia, Mr Amiruddin, and Mr Lukman. The implementation of oop (object oriented programming) in building an e-commerce website. In 1st International Conference on Advanced Multidisciplinary Research (ICAMR 2018). Atlantis Press, 2019.

[42] Ovidiu Constantin Novac, Mihaela Novac, Cornelia Gordan, Tamas Berczes, and Gyöngyi Bujdosó. Comparative study of google android, apple ios and microsoft windows phone mobile operating systems. In 2017 14th International Conference on Engineering of Modern Electric Systems (EMES), pages 154–159. IEEE, 2017.

[43] G. A. S. Oliveira and R. Bonacin. A method for teaching object-oriented programming with digital modeling. In 2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT), pages 233–237, July 2018. doi: 10.1109/ICALT.2018.00060.

[44] Mikael Olsson. C++ quick syntax reference. In C++ Quick Syntax Reference, pages 41–44. Springer, 2013.

[45] MS Jorge L Ortega-Arjona. Curb your objects! an orthodox form for c# classes.

[46] Norbert Pataki and Zoltán Porkoláb. Extension of iterator traits in the c++ standard template library. In 2011 Federated Conference on Computer Science and Information Systems (FedCSIS), pages 911–914. IEEE, 2011.

[47] Indrajeet Patil. Test and effect size details. online, December 2020. URL `https://cran.r-project.org/web/packages/statsExpressions/vignettes/stats_details.html`. visited 28 december.

[48] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y. Guéhéneuc. Towards understanding interactive debugging. In 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), pages 152–163, Aug 2016. doi: 10.1109/QRS.2016.27.

[49] Dmitrii Rassokhin. The c++ programming language in cheminformatics and computational chemistry. Journal of Cheminformatics, 12(1):10, 2020.

[50] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 485–495, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337280`.

[51] Bjarne Stroustrup. The C++ programming language. Pearson Education, 2013.

[52] Jun-Ming Su and Feng-Yuan Hsu. Building a visualized learning tool to facilitate the concept learning of object-oriented programming. In 2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), pages 516–520. IEEE, 2017.

[53] A. Treffer and M. Uflacker. Back-in-time debugging in heterogeneous software stacks. In 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 183–190, Oct 2017. doi: 10.1109/ISSREW.2017.62.

[54] József Udvaros and Miklós Gubán. Demonstration the class, object and inheritance concepts by software. Acta Didactica Napocensia, 9(1):23–34, 2016.

[55] Murat Pasa Uysal. The effects of objects-first and objects-late methods on achievements of oop learners. Journal of Software Engineering and Applications, 5(10):816, 2012.

[56] Karl L. Wuensch. Nonparametric effect size estimators. http://core.ecu.edu/psyc/wuenschk/docs30/Nonparametric-EffectSize.pdf, July 2020. visited 28-dec-2020.

[57] Stelios Xinogalos. Object-oriented design and programming: An investigation of novices&rsquo; conceptions on objects and classes. Trans. Comput. Educ., 15(3):13:1–13:21, July 2015. ISSN 1946-6226. doi: 10.1145/2700519. URL `http://doi.acm.org/10.1145/2700519`.

[58] J. Yang, Y. Lee, D. Hicks, and K. H. Chang. Enhancing object-oriented programming education using static and dynamic visualization. In 2015 IEEE Frontiers in Education Conference (FIE), pages 1–5, Oct 2015. doi: 10.1109/FIE.2015.7344152.

[59] Andreas Zeller and Dorothea Lütkehaus. Ddd—a free graphical front-end for unix debuggers. ACM Sigplan Notices, 31(1):22–27, 1996.

Appendix A

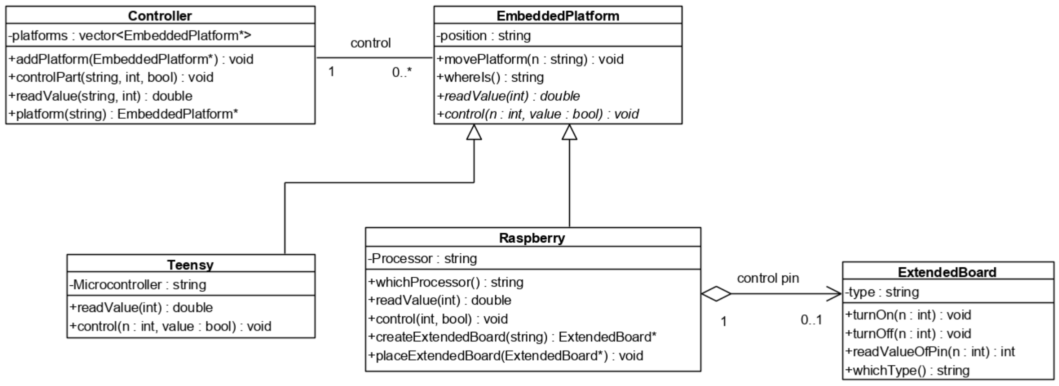
Glossary

BSA	Binding Study Advice
DDD	Data Display Debugger
EMBED	Embedded Programming
OOMOD	Object Oriented Modelling
OOP	Object Oriented Programming
OOPR1	Introduction object oriented programming
OOPR2	Object oriented programming part II
R_{it}	Rasch UnIT
SWADP	Software Architecture and Design Patterns

Appendix B

Exam 2019, Introduction Object-Oriented Programming (translated to English)

In the class diagram below, the Controller class controls various embedded platforms. Some platforms also have a setup print. Each platform can read a value from a certain pin, this pin could also be an analogue input. Each platform can also put a certain value on a PIN.



Controller
-platforms : vector<EmbeddedPlatform*>
+addPlatform(EmbeddedPlatform*) : void
+controlPart(string, int, bool) : void
+readValue(string, int) : double
+platform(string) : EmbeddedPlatform*
control
1
0..*
EmbeddedPlatform
-position : string
+movePlatform(n : string) : void
+whereIs() : string
+readValue(int) : double
+control(n : int, value : bool) : void
Teensy
-Microcontroller : string
+readValue(int) : double
+control(n : int, value : bool) : void
Raspberry
-Processor : string
+whichProcessor() : string
+readValue(int) : double
+control(int, bool) : void
+createExtendedBoard(string) : ExtendedBoard*
+placeExtendedBoard(ExtendedBoard*) : void
control pin
1
0..1
ExtendedBoard
-type : string
+turnOn(n : int) : void
+turnOff(n : int) : void
+readValueOfPin(n : int) : int
+whichType() : string

Figure B.1: Class diagram of questions A-K

```
int main() {

    Controller ctr;
    AlgController<int> algctr;
    ExtendedBoard opz("temperature");            //extended board type temperature
    Raspberry pi("ARM Cortex A7","living room"); //raspberry with processor ARM Cortex A7, will be placed in living room
    Teensy ts("ARM CortexM4","gang");          //Teensy with processor ARM Cortex M4, will be placed in corridor
    pi.placeExtendedBoard(&opz);                 //place opz on the pi
    ctr.addPlatform(&pi);                  // add both platforms
    ctr.addPlatform(&ts);                  //

    Raspberry reserv_pi(pi);               //Make a completely copy  of the pi
    reserv_pi.control(1,true);
    cout<< "pi "<<pi.whichProcessor()<<"  reserv "<<reserv_pi.whichProcessor()<<endl;

    double x=ctr.readValue("living room",2);    //read the value from the platform in the living room on pin 2
    if(x >2.8)
        ctr.controlPart("living room",1,true);  // Pin 1 true of living room platform

    EmbeddedPlatform* i=ctr.platform("living room");   //return a pinter to the platform in the living room.

    if(i)                                            // is there a platform in the living room.
       double x1=i->readValue(2);


    return 0;
}
```

Figure B.2: The main code

```
class Controller {

  public:
     void addPlatform(EmbeddedPlatform* pf);
     void controlPart(string positie, int nr, bool value);
     double readValue(string,int n)const;
     EmbeddedPlatform* platform(string) const;

  private:
     vector<EmbeddedPlatform*> platforms;
 };
```

Figure B.3: Class Controller

```
class EmbeddedPlatform {

private:
    string position;

public:
    void movePlatform(string n);

    string whereIs() const;

    virtual double readValue(int)const=0;

    virtual void control(int n, bool value) = 0;
    virtual ~EmbeddedPlatform(){}
    EmbeddedPlatform(string);
};
```

(a)

```
class Teensy : public start::EmbeddedPlatform {

public:
    Teensy(string,string);

    double readValue(int)const;

    void control(int n, bool value);
private:
    string Microcontroller;
};
```

(b)

```
class Raspberry : public start::EmbeddedPlatform {

private:
    string Processor;
    ExtendedBoard* control_device;
public:
    Raspberry(string,string);
    Raspberry(const Raspberry&);
    string whichProcessor() const;
    double readValue(int)const;
    void control(int n, bool value);
    void placeExtendedBoard(ExtendedBoard*);
    ExtendedBoard* createExtendedBoard(string);

};
```

(c)

```
class ExtendedBoard {

private:
    string type;

public:
    ExtendedBoard(string);
    void turnOn(int n);
    void turnOff(int n);
    int readValueOfPin(int n) const;
    string whichType() const;
};
```

(d)

Figure B.4: My composed figure

A) What is the consequence in this example if `virtual` of the function `virtual double readValue(int) const = 0;` of the class **EmbeddedPlatform** is omitted.

5P

B) In this example, what is the result if `= 0` after the function `virtual double readValue( int ) const = 0;` of the class **EmbeddedPlatform** is omitted.

5P

C) Is there a ”memory leak” in the code of the main routine, give a short explanation.

5P

D) Is the slicing problem mentioned in the code of the main routine, give a short explanation.

5P

E) Write down the implementation (program code) of constructor of the class **Embedded Platform**.

5P

F) The method `void addPlatform(EmbeddedPlatform *);` of the class **Controller** adds an embedded platform to the vector `platforms`. Write down the implementation (program code) of the method: `void add PlatformToe (EmbeddedPlatform *);`

5P

G) The method `double readValue (string, int) const;` of the class **Controller** returns the value of the platform that is located on the given position and the pin in question. Write down the implementation (program code) of the method:
`double readValue (string, int) const;`

10P

H) Give the implementation (program code) of constructor of the class **Raspberry** 10P

I) On the line `Raspberry reserv_pi(pi);` of the main program is created a new raspberry platform with the same features as the platform *pi*. Write down what goes wrong here and enter the code to correct this error.

10P

J) The method `void control(int n, bool v);` of the Raspberry class, turn on or off pin `n` of the extended board, if this is present. If no extended board is present, nothing happens. Write down the implementation (program code) of the method:
`void control (int n, bool v);`

10P

K) If the class **Controller** is generalized so that it also could be used to control other types of objects e.g. class **Desktop**. The Class **Controller** has to be made a template class. Make from the class **Controller** a template class (adjust only the class declaration, the names of the methods may remain the same). Show also how the template class has to be created around objects of e.g. desktop.

10P

L) Given the sequence diagram below which calls the method
`ExtendedBoard* createExtendedBoard(string);` of the class **Raspberry**.

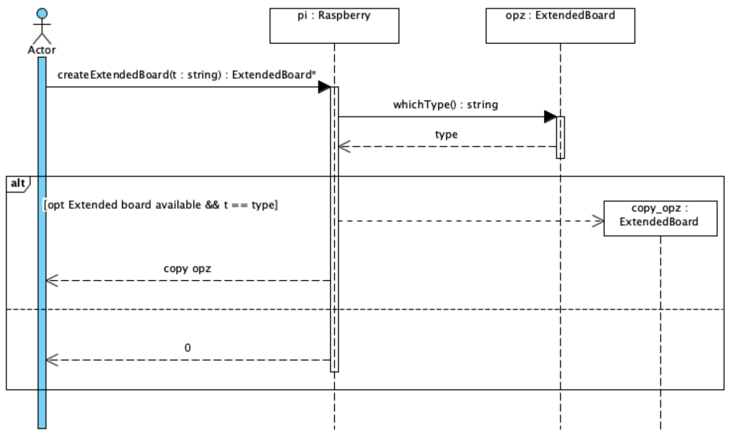


Figure B.5: Sequence diagram of method `createExtendedBoard`.

Write down the implementation (program code) of the method:
`ExtendedBoard* createExtendedBoard(string);`

10P

Appendix C

Exam 2020, Introduction Object-Oriented Programming (translated to English)

Assignment 1

In the class diagram below, the class IoTUnit checks the IoT_devices. An IoT_device checks or measures a particular unit. For example, the barometer measures air pressure using an air pressure sensor. This air pressure sensor can be set so that the measured value in hectoPascal or is displayed in bar; furthermore, the accuracy can be set to a 16 bits or a 32 bit value. The thermometer measures the temperature using a temperature sensor. This temperature sensor can display the measured values in degrees Celsius, Kelvin or Fahrenheit; further, and the accuracy can show be set to a 16 bit or a 32 bit value.

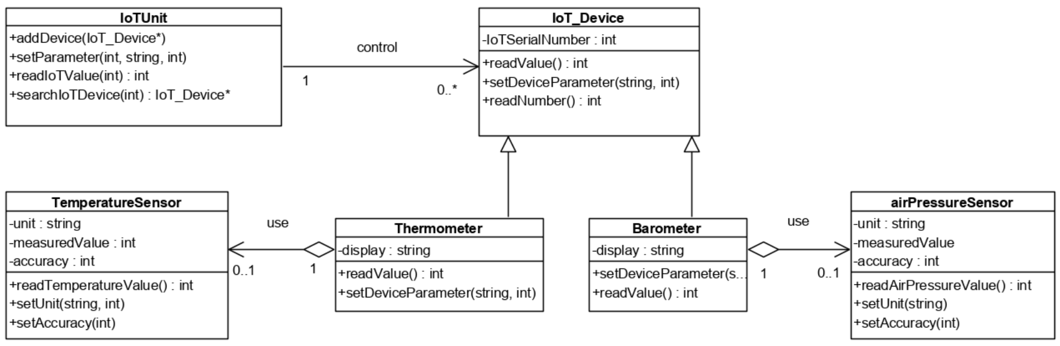


Figure C.1: Class diagram of assignment 1

```
int main() {

    IoTUnit building1;

    //Air pressure sensor is set at 16 bit accurucy unit = bar

    AirPressureSensor lds1(16,"bar");
    Barometer bm1(1234,"analoog",&lds1);
    AirPressureSensor lds2(16,"bar");
    Barometer bm2(5678,"digitaal",&lds2);

    //add barometers
    building1.addDevice(&bm1);
    building1.addDevice(&bm2);

    //Barometer 1, is set with hectoPacal as a unit and a 32 bit accuracy
    building1.setParameter(1234,"hectoPascal",32);
    int waardeBm1=building1.readIoTValue(1234); //read out the air pressure of barometer 1.

    IoT_Device *dv1=&bm1;
    int value1=dv1->readValue();

    IoT_Device dv2=bm1;
    int value2=dv2.readValue();


};
```

```
class IoTUnit {

    public:
        void addDevice(IoT_Device* iotD);
        void setParameter(int IoTnr, string p, int waarde);
        int readIoTValue(int w) const;
        IoT_Device* searchIoTDevice(int nr) const;

    private:
        vector<IoT_Device*> controller;
};
```

```
class IoT_Device {

        private:
            int IoTSerialNumber;

        public:
            IoT_Device(int);
            virtual ~IoT_Device();
            virtual int readValue() const;
            virtual void setDeviceParameter(string wat, int waarde);
            int readNumber() const;
};
```

```cpp
class Barometer : public IoT_Device {

        private:
            string display;
            AirPressureSensor* airsens;

        public:
            Barometer(int,string,AirPressureSensor*);
            void setDeviceParameter(string wat, int waarde);
            int readValue() const;
};
```

```cpp
class Thermometer : public IoT_Device {

        private:
            string display;
            TemperatuurSensor* temps;

        public:
            Thermometer(int,string,TemperatuurSensor*);
            int readValue() const;
            void setDeviceParameter(string wat, int waarde);
};
```

```cpp
class AirPressureSensor {

        private:
            int accuracy;
            string unit;
            int measuredValue;

        public:
            AirPressureSensor(int,string);
            int readAirPressureValue();
            void setMeasurement(string eenheid);
            void setAccurance(int nw);
};
```

A What is the consequence in this example if `virtual` of the method `virtual double readValue(int);` of the class **IoT_Device** is omitted.

5P

B What is the consequence in this example if `= 0` after the method `virtual double readValue( int ) const;` of the class **IoT_Device** is placed.

5P

C Is there a "memory leak" in the code of the main routine, give a short explanation.

5P

D Is the slicing problem mentioned in the code of the main routine, give a short explanation.

5P

E Write down the implementation (program code) of constructor of the class **IoT_Device**.

5P

F The method `void addDevice(IoT_Device *);` of the class **IoTUnit** adds an IoT_device to the
vector `controller`.
Write down the implementation (program code) of the method:
`void addDevice(IoT_Device *);`

5P

G The method `double readIoTValue (int) const;` of the class **IoTUnit** returns the value of the platform that is located on the given position and the pin in question. Write down the implementation (program code) of the method:
`double readIoTValue (string) const;`

10P

H Give the implementation (program code) of the constructor of the class **Barometer**

10P

I The method `void setDeviceParameter(string w, int value);` of the class **Barometer**, if the sensor is present, set the unit and accuracy of the sensor. If no sensor is present, nothing happens. Write down the implementation (program code) of the method:
`void setDeviceParameter(string w, int value);`

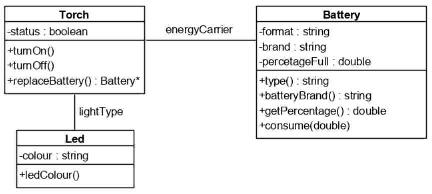
10P

J If the class **IoTUnit** is generalized so that it also could be used to control other types of objects e.g. class **Robot**. The Class **IoTUnit** has to be made a template class. Make from the class **IoTUnit** a template class (adjust only the class declaration, the names of the methods may remain the same). Show also how the template class has to be created around objects of e.g. robot.

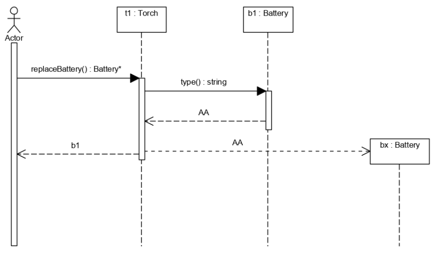
10P

Assignment 2

Examine the class and sequence diagram below and the code.



(a) Class diagram of assignment 2.



(b) Sequence diagram of assignment 2

Figure C.2: Assignment 2, Change of a battery

```
        class Torch
        {
                public:
                    Torch(Battery*,Led*);
                    void turnOn();
                void turnOff();
                Battery* replaceBattery();

                private:
                Battery* bat;
                Led* ld;
                bool status;
        };
```

```
    Led ledx("red");
    Battery b("AA","Duracel");
    Torch light(&b,&ledx);
    Battery* old=light.replaceBattery();
```

A The method `Battery* replaceBattery();` of the class **Battery** creates a new battery and returns the old battery, as shown is in the sequence diagram.
Write down the implementation (program code) of the method:
`Battery* replaceBattery();`

B Implement the copy constructor of the class torch, so that with the statement `Torch reserveLight(light);` the reserveLight object is the same as the object light but with different LEDs and batteries.

Appendix D

Exam 2019, Object-Oriented Programming, part II (translated to English)

In the UML diagram below, a system is shown in which several type of switches, different types of lighting can be turned on and off. Both the switches and the lighting must be registered with the controller, indicating in which room they will be placed.

```
int main()
{
        cout<<"help"<<endl;
        Controller home;
        TemperatureSensor ts1;
        ts1.takeMeasure();
        Timer t1(100);
        Lamp lmp1(35,"E27");
        Led ld1("White",10);
        Led ld2("Yellow",5);
        Thermostat tm1("L␣&␣B",&ts1);
        ManualSwitch hs1("Brown");
        ManualSwitch hs2("Yellow");
        Display d2;
        hs1.connectDisplay(&d2);
        hs1.connectController(&home);

        tm1.connectController(&home);

        home.addSwitch(&tm1,"livingroom");
        home.addSwitch(&hs1,"shed");
        home.addLightning("livingroom",&ld2);
        home.addLightning("shed",&lmp1);
        home.addLightning("shed2",&ld1);

        hs1.push();

        return 0;
}
```

Assignment 1

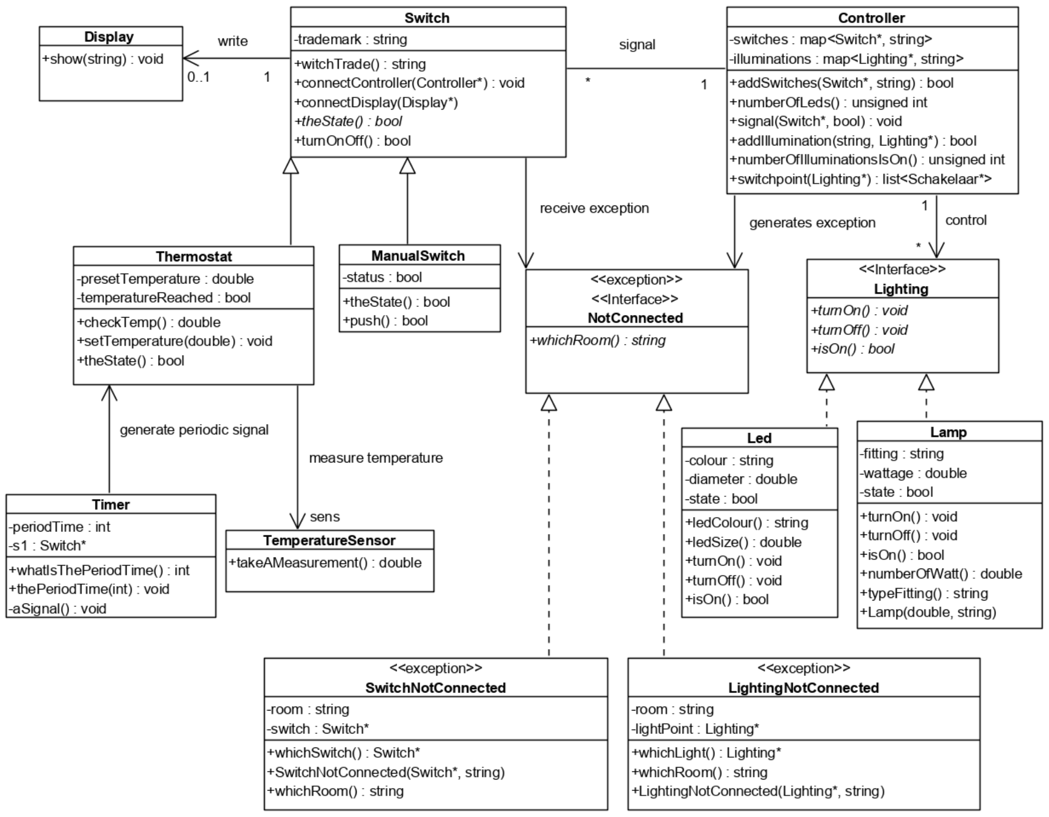


Figure D.1: Class diagram of the exam March 2019

The class **ManualSwitch** is a switch. The manual switch keeps track of the condition (contact or no contact). When the method `bool push()` is called, the controller is notified that the button has been pressed. This is done by the method `bool turnOnOff()`.

Make the class **ManualSwitch** and indicate exactly what will be in the header file and what will be in the implementation file.

12P

Assignment 2

The method `double checkTemp();` the class **Thermostat** performs a temperature measurement (measures whether the temperature has already been reached) and returns the current temperature measured by the sensor.
Make the method `double checkTemp();`

5P

Assignment 3

The method `bool turnOnOff();` of the class **Switch**, if connected to a controller, calls the method

`void signal(Switch*,bool);` of the class **Controller**. The method `signal` can generate a `SwitchNot-connected` exception if the switch is not known to the controller. If the exception occurs, the room where it is located will be shown on the display.
Make the method `double checkTemp();`

20P

Assignment 4

The class **lighting** is an interface.
Give the declaration of the class **lighting**.

6P

Assignment 5

The method `bool addSwitches(Switch* s, string place);` of the class **Controller** adds a switch with the given place. If there is already a switch in the location, a false will be returned.
Make the method: `bool addSwitches(Switch* s, string place);`

6P

Assignment 6

The method `void signal(Switch* s, bool b);` of the class **Controller** turns the lighting(s) on/off in the same room as where the switch is located. If the switch is not known to the controller, a `SwitchNot-connected` exception is generated with the space ”unknown”. If the Switch is known to the controller, but no lighting is present in the room, a `SwitchNot-Connected` Exception is generated with as room, the room where the Switch is located.
Make the method: `void signal(Switch* s, bool b);`

17P

Assignment 7

The method `unsigned int numberOfIlluminationsIsOn()const;` of the class **Controller** returns the number of lights that are on.
Make the method: `unsigned int numberOfIlluminationsIsOn()const;`

6P

Assignment 8

The method `unsigned int numberOfLeds()const;` of the class **Controller** returns the number of connected LEDs in the controller.
Make the method: `unsigned int numberOfLeds()const;`

8P

Appendix E

Exam 2020, Object-Oriented Programming, part II (translated to English)

(Because this exam was taken digitally, it consists of four parts and general information.)

General Information

The exam starts at 09:00 and consists of a general part (class diagram and the main) and three separate parts with two or four questions each.

Please upload a **separate** word or pdf file for each part.

- The general part will be visible during the entire examination.9:00 ... 11:00
- Questions 1 and 2 are available and can be uploaded in word or pdf from 09:05 ... 09:30.
- Questions 3 and 4 are available and can be uploaded in word or pdf from 09:30 ... 10:05.
- Questions 5, 6, 7 and 8 are available and can be uploaded in Word or pdf from 10:05 ... 10:50.

Too late is too late, so please upload before the time runs out.

For the variable within a method, take original names, starting with the first two letters of your surname and they end with the last two digits of your study number. So:
John van de Object: has study number 12389.

~~int a;~~ is wrong
int obj_iter89; is okay
Don’t forget to **refresh** the page.

General part

The class diagram below shows a system where different products can be scanned through different types of cash registers. In the product processor, the price of each product is stated, as well as the cash registers and whether or not a cash register is open. If it has been paid for, a gate opens at the self-service cash register and the customer can pass through it.

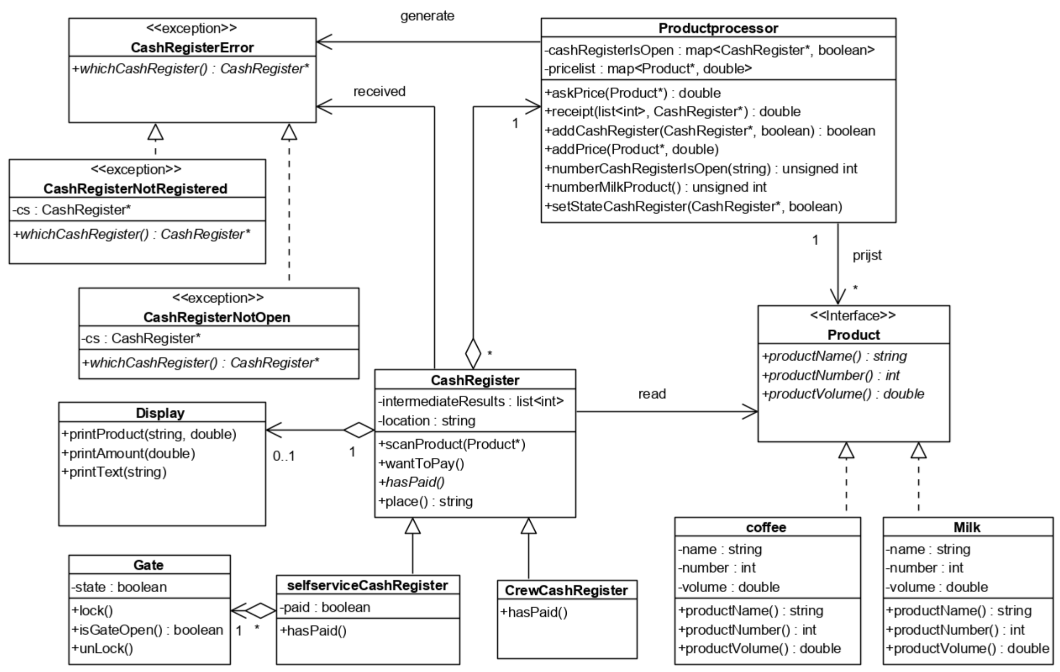


Figure E.1: Class diagram of the exam March 2020

```
int main()
{
        cout<<"begin"<<endl;

        Productprocessor ob_processor89;
        Milk ob_ml1_89("Campina␣halfFull",01001,1000);
        Coffee ob_kf1_89("Peeze␣Mex␣dark",02001,250);
        Milk ob_ml2_89("Campina␣full",01002,1000);
        Milk ob_ml3_89("AH",01003,1000);
        ob_processor89.addPrice(&ob_ml1_89,1.65);
        ob_processor89.addPrice(&ob_ml2_89,1.75);
        ob_processor89.addPrice(&ob_ml3_89,0.65);
        ob_processor89.addPrice(&ob_kf1_89,8.45);
        Gate ob_poort89;
        Gate ob_poort2_89;
        Display ob_d1_89;
        CashRegister* ob_selfs1_89=new SelfServiceCashRegister(&ob_poort89,
                                    &ob_d1_89,&ob_processor89,"Delft");
        ob_processor89.addCashregister(ob_selfs1_89,true);
        CashRegister* ob_selfs2_89=new SelfServiceCashRegister(&ob_poort2_89,
                                    &ob_d1_89,&ob_processor89,"Delft");
        ob_selfs1_89->scanProduct(&ob_ml1_89);
        ob_selfs1_89->scanProduct(&ob_ml1_89);
        ob_selfs1_89->scanProduct(&ob_kf1_89);
        ob_selfs1_89->scanProduct(&ob_kf1_89);
        ob_selfs1_89->wantToPay();

        ob_selfs2_89->scanProduct(&ob_ml2_89);
        ob_selfs2_89->scanProduct(&ob_ml3_89);
        ob_selfs2_89->wantToPay();

        delete ob_selfs1_89;
        delete ob_selfs2_89;
        return 0;
}
```

For the variable within a method, take original names, starting with the first two letters of your surname and they end with the last two digits of your study number. So:
John van de Object: has study number 12389.

~~int a;~~ is wrong
int obj_iter89; is okay

Assignment 1

The class **SelfServiceCashRegister** is a CashRegister. At the self service cash register, a gate is unlocked as soon as payment has been made, so that the customer can go through it. The method *`void hasPaid();`* first indicates that payment has been made, then the gate is unlocked, after which it is again indicated that payment has not been made.

Make the class **SelfServiceCashRegister** and indicate exactly what will be in the header file and what will be in the implementation file.

12P

Assignment 2

The class **Product** is an interface.
Give the declaration of the class **Product**.

6P

For the variable within a method, take original names, starting with the first two letters of your surname and they end with the last two digits of your study number. So:
John van de Object: has study number 12389.

~~int a;~~ is wrong
int obj_iter89; is okay

Assignment 3

The method `void scanProduct(Product*);` from the class CashRegister scans a product. The number of the product is stored in the list intermediateResults. Make the method:`void scanProduct(Product*);`

5P

Assignment 4

The method `void wantToPay();` of the class **CashRegister** is invoked when payment is due. First, the final amount is called up (the method `receipt` of the class **ProductProcessor** is called) which returns the final amount. After this, it is indicated that payment has been done, so that the gate can be opened at the self-service cash register or the cashier can ask for stamps at the crew cash register.
The method `receipt` can also generate an exception if the cash register does not appear (CashRegisterNotRegistered) or the cash register is not open (CashRegisterNotOpen). If an exception is received, the display will show which exception is received and the location of the cash register given. Make the method: `void wantToPay();`

20P

For the variable within a method, take original names, starting with the first two letters of your surname and they end with the last two digits of your study number. So:
John van de Object: has study number 12389.

~~int a;~~ is wrong
int obj_iter89; is okay

Assignment 5

The method `bool addCashRegister(CashRegister*, bool);` of the class **Productprocessor** adds a cash register with the given place. If there is already a cash register, a false will be returned.
Make the method: `bool addCashRegister(CashRegister*, bool);`

6P

Assignment 6

The method `unsigned int numberCashRegisterIsOpen(string) const;` of the class **Productprocessor** returns the number of cash registers which have the given string as location.
Make the method: `unsigned int numberCashRegisterIsOpen(string) const;`

6P

Assignment 7

The method `unsigned int numberMilkProduct() const;` of the class **Productprocessor** returns the number of milk products in the map pricelist
Make the method: `unsigned int numberMilkProduct() const;`

8P

Assignment 8

The method `double receipt(list<int> productnumbers,CashRegister* );` of the class **Productprocessor** calculates the total price of the given production numbers. The first step is to check whether the cash register is present and open. The cash register does not occur then a `CashRegisterNotRegistered` exception will be generated. If the cashier does occur but it is not open, a `CashRegisterNotOpen` exception is generated.
If the cash register does occur and it is open, the product and the corresponding price will be searched for each given product number. The total price is returned.
Make the method:
`double receipt(list<int> productnumbers,CashRegister* );`

17P

Appendix F

The results of the exam January 2020.

			Opgave 1										Opgave 2	
	Score	total	A	B	C	D	E	F	G	H	I	J	A	B
maximum number of points	10 final	90	5	5	5	5	5	5	10	10	10	10	10	10
total number of points		4770	265	265	265	265	265	265	530	530	530	370	530	530
total number of points by students		2887	132	142	179	58	258	227	325	413	380	273	276	224
avarage	6.45	54.47	2.49	2.68	3.38	1.09	4.87	4.28	6.13	7.79	7.17	5.15	5.21	4.23
variance	4.65	465.37	5.02	2.80	5.51	4.24	0.19	2.51	14.39	10.63	14.53	13.94	12.21	18.37
standard deviation	2.2	21.6	2.2	1.7	2.3	2.1	0.4	1.6	3.8	3.3	3.8	3.7	3.5	4.3
high score	9.7	87.0	5.0	5.0	5.0	5.0	5.0	5.0	10.0	10.0	10.0	10.0	10.0	10.0
M/Mmax			0.5	0.5	0.7	0.2	0.97	0.9	0.6	0.8	0.7	0.5	0.5	0.4
r_{it}			0.5	0.7	0.5	0.3	0.3	0.6	0.8	0.8	0.9	0.7	0.7	0.6
Cronbach's alpha		0.85												

Table F.1: Analyse of the results of exam January 2020

Appendix G

Timetable of the course Introduction Object Oriented programming

week	subjects
1	Repeating of the course object modelling from the previous term. First principles of a class. Compare class with structs in program language C.
2	short repeating last week, elaborate on the constructor, composition, references
3	short repeating last week, aggregation / association, inheritance, slicing problem
4	short repeating last week, Overloading, templates and vector.
5	short repeating last week, allocate dynamic memory (new), destructor and why own destructor declare, copy constructor and operator = and why they have to be declared instead of using the default methods
6	static members, namespace and operator overloading.
7	discuss the test exam

Table G.1: Subjects of the old OO course.