Master thesis
Computer Science

Radboud University

Designing the Teaching Pattern: a structured approach to teach design patterns

The design and evaluation of a structured teaching activity to teach students design patterns

Author:
Ruben Holubek
s1006591

First supervisor/assessor:
Sjaak Smetsers
`sjaak.smetsers@ru.nl`

Second assessor:
Erik Barendsen
`erik.barendsen@ru.nl`

December 20, 2022

Abstract

Design patterns are an essential topic in Object-oriented programming. However, students find this a challenging topic, mainly due to the abstract nature of design patterns and students do not see the benefits of applying design patterns, simply because they do not have enough experience with programming yet. Additionally, learning activities about design patterns often focus on implementing patterns, but the application of design patterns in practice is often skipped. As a result, it is quite challenging to effectively teach design patterns. In the literature, there are some guidelines provided to teach design patterns, but no concrete structure is provided yet. Therefore, we aim to design a lecture with a concrete structure for design patterns which is an extension of these existing guidelines.

In our research, we designed and performed a learning activity to teach design patterns and inspected the students' understanding of design patterns afterwards. For this design, we looked at the problems and solutions for teaching (Object-oriented) programming presented in the literature as well as these existing guidelines. The learning activity consisted of a traditional lecture and an assignment. The lecture was made with a specific structure, which included formative assessment via an online quiz and live coding sessions. Additionally, it focused on the implementation element of design patterns, as well as the application element. Afterwards, we collected data from the students, ranging from their experience in the lecture to their handed-in exam.

Our results have shown that students were very positive about the lecture: the lecture was interactive and they appreciated the online quiz. In the assignment, students could apply design patterns with little problems and had a high-level understanding of the UML of design patterns. However, the students had problems with the details of the implementation of design patterns, namely the bodies of the functions in the pattern, the relations between functions and the code needed to actually use these design patterns.

Contents

Chapter 1

Introduction

1.1 Design patterns

Object-oriented programming is one of the most used programming styles and is present in most of the Computing Science curricula. One of the most fundamental concepts of Object-oriented programming is structuring programs with the help of classes and other datastructures. Therefore, design patterns are often closely related to Object-oriented programming and will certainly be discussed at one point in a Computing Science study (Xinogalos, 2015).

Design patterns are typical solutions to commonly occurring problems in software design, mainly beneficial for the maintenance of the program and the extensibility (Shosse, 2022). They can be seen as blueprints that programmers can customize to solve their specific problem. These different design patterns can be roughly categorised into three classes: creational patterns, structural patterns and behavioral patterns. Creational patterns are mechanisms that easily and efficiently create objects, which increases the flexibility and reusability of the code and isolates the code related to the creations of objects of the programs. Examples are the Factory pattern (Explanation) or the Singleton pattern (Explanation). Structural patterns focus more on assembling objects and classes into larger structures, which are easily maintained and extended, of which the Decorator pattern is widely known (Explanation) or the usage of an Adapter (Explanation). Behavioral patterns are the patterns closest to algorithms, because these focus on the responsibilities between objects and classes and interactions resulting in a system of objects which can easily be extended and maintained. Examples are the Observer pattern (Explanation), Iterator pattern (Explanation) or the Visitor pattern (Explanation).

Design patterns are sometimes compared to algorithms, but in essence design patterns are quite different: algorithms are specific actions that can be performed to solve a problem, while a design pattern provides a high-level description of a solution, of which the details are not provided and should be implemented by the programmers. This is also the reason why design patterns differ from libraries, as they are not pre-written code (Shosse, 2022). However, this is the key reason why students often struggle with the concepts of design patterns, as the students are not (yet) used to think about these high-level descriptions at that point of their study. Therefore, even though design patterns are essential for Object-oriented programming, students are initially struggling with these concepts which results in a educational challenge (Azimullah, An, & Denny, 2020).

1.2 Problems related to teaching Object-oriented programming

There are multiple reasons why students find the topic of design patterns difficult, as pointed out in literature. One of the reasons is that design patterns are very abstract in nature, which is difficult to immediately grasp for the students (Silva, Schots, & Duboc, 2019). Even though the design patterns are abstract, they are still dynamic concepts; there are a lot of interactions between the different Classes and Objects. However, these patterns are often presented on slides, which is relatively static medium to use, so the dynamic nature of the patterns is more challenging to present (Azimullah et al., 2020). Furthermore, students do not understand the utility of the design patterns yet (Mello Fonseca, Bezerra da Silva, & Silveira Mendonça, 2019), which also becomes clear from the fact that the students are unsure when to apply patterns and often incorrectly use or overuse design patterns (Lotlikar & Kussmaul, 2022). This latter point is also related to the lectures about design patterns that are often focusing on the implementation of design patterns instead of actually using them (Lotlikar & Kussmaul, 2022).

Some of these problems related to teaching design patterns are similar to observations seen in teaching Object-oriented programming. For example, Object-oriented programming is often seen as a challenging style of programming, due to its abstract nature and the fact that it is often given as a second programming course for the students (Medeiros, Ramalho, & Falcão, 2019). As a result, students are still focused on the functionality instead of the structure of a program, so it is difficult of the students to make this change in mindset (Chibizova, 2018). Related to this, students simply do not have enough experience in programming to actually see and understand the presented problems and solutions (Gutierrez, Guerrero, & López-Ospina, 2022).

The literature also proposed some solutions for these challenges. Unfortunately, there is little research conducted for tackling the problems seen in teaching design patterns; Lotlikar and Kussmaul (2022) provided some guidelines for teaching design patterns, but there was no concrete teaching method we could find. There was more research done on teaching Object-oriented programming, which are discussed in more detail in chapter 2, including the research done related to teaching design patterns. One solution we often saw was the usage of alternative teaching methods instead of the traditional lecture.

1.3 Traditional lecture versus alternative teaching methods

There are three alternative teaching methods that are effectively used in combination with (Object-oriented) programming: flipped classroom, blended classroom and problem-based learning (PBL). The idea of flipped classroom is that the students should learn the new material themselves before the meeting with the teacher, which is in return only focused on applying the learned knowledge and practicing (Sharma, Biros, Ayyalasomayajula, & Dalal, 2020a). A blended classroom is in essence a less-restricted flipped classroom, as the meeting with the teacher starts with a small lecture which recaps the material that the students already looked at. Afterwards, the meeting is resumed with the session where the students practice with the material (Alammary, 2019). The idea of Problem-based learning (PBL) is that the teacher does not give any lectures at all, but the students learn independently by building programs for authentic and real-world problems. When the students face a problem, they google the solution or ask the teacher, so they essentially learn by doing, instead of being lectured (Chis, Moldovan, Murphy, & Muntean, 2018).

One thing these alternative teaching methods have in common is that they let the students practice with the material most of the time, instead of using the meetings with the teacher as a traditional lecture. These researches have shown that these alternative methods are more effective, because programming is mostly a skill learned by doing. However, in practice, we still see that the traditional lecture setting is often used instead of these "better" alternative teaching methods. One reason is that teachers often do not have the time to completely change their structure of their courses, which often leads to simply keeping the used slides for years and the lecture setting (Alaagib, 2019). More interestingly, we see that these alternative teaching methods are the most effective in smaller groups and in a physical setting. However, we see that the groups in the courses only get bigger, thus such an alternative teaching methods is quite challenging to effectively apply here (Weiss, 2020). Additionally, we see a shift towards online education, especially after the changes related to Covid, and these alternative methods are not created to be applied in an online setting (Hlescu, Birlescu, Hanganu, Manoilescu, & Ioan, 2020). For this reason, the traditional lecture setting is still widely used in practice.

1.4 Overview of this research

The main goal of this research is to identify the students' understanding of design patterns and how a learning activity can contribute to this. This will be done by designing and performing a learning activity about design patterns and afterwards, collecting data from students to see their understanding of design patterns. Due to the setting in which we conduct this research, the learning activity has to be given as a traditional lecture. Therefore, we cannot use any alternative teaching method, but it still makes this research relevant for traditional lectures, which are widely used in practice (Alaagib, 2019).

In the literature, problems related to teaching design patterns and teaching (Object-oriented) programming are discussed as well as potential solutions to these problems and several guidelines for effective lectures about design patterns. However, there is no concrete structure for a lecture about design patterns in the literature and therefore, we aim to design such a structured lecture. The goal is to tackle these problems by combining these guidelines and effective approaches for teaching (Object-oriented) programming, like live coding and formative assessment via an online quiz. Additionally, we are interested whether this approach is viable for bigger groups and an online environment.

Afterwards, we want to get some insight in the learning effects of the lecture and the students' perception and experience of the given lecture. Furthermore, we want to see if the students understood the design patterns, which concepts are still unclear and which problems they are facing. These questions will be answered by looking at different data sources, for example the students' work and questionnaires filled in by the students. By combining all these findings, we can characterise the students' understanding of design patterns and see how a learning activity can contribute to this, by looking at our case study.

1.5 Structure of this thesis

This thesis is structured as follows: in chapter 2, other literature is discussed related to our research; problems and solutions for teaching (Object-oriented) programming and design patterns and ideas for effectively evaluating a learning activity. In chapter 3 the research (sub)questions are stated. In chapter 4, the setting is described in which this research is conducted, as well as the changes we made to the lecture and assignment. In chapter 5, our methods are discussed for the collected data and the results of these are discussed in chapter 6. Lastly, the conclusions, thus the answers to the research questions, are presented in chapter 7 and other observations, results and a next iteration of our research are discussed in chapter 8. In the appendices, different documents can be found, including the slides used in the lecture and the programs used in the learning activity.

Chapter 2

Theoretical Framework

In this chapter, we discuss the current knowledge related to teaching programming. Within the first three sections, we discuss the problems and solutions in the literature related to teaching programming in general, Object-oriented programming and design patterns. Note that these observations are related to learning activities in a traditional lecture setting. In the fourth section, we look at the different methods used in the literature to see the effects of a learning activity and to inspect the students when they are working on a program.

2.1 Teaching Programming

A lot of research has been done related to the problems students experience when learning to program. The lectures within these courses can have a big impact on these experiences, but the literature also pointed out several issues with the most-common teaching method: a traditional lecture (Sharma et al., 2020a). Therefore, there has been a few suggestions in the literature to make these traditional lectures more effective. This section discusses problems related to teaching programming to students and suggestions how to improve this in a traditional lecture setting.

2.1.1 Problems

Students get overwhelmed with examples

The main difficulty for students when they learn to code, is to get used to the level of abstractness that comes with programming (Medeiros et al., 2019). To tackle this, especially in programming courses for beginners, teachers show a lot of concrete examples of different programs, which is indeed a good solution. However, these examples are often too difficult or too complicated for the students to fully, or even partially, understand. As a result, students get overwhelmed and are struggling to follow the remainder of the lecture

(Sharma, Biros, Ayyalasomayajula, & Dalal, 2020b). Additionally, it does not help that in most lectures a clear gradual buildup is missing in the structure of the slides, such that students can easily follow along (Robins, Rountree, & Rountree, 2003).

Code is presented on slides

As in the traditional setting, complete and correct code is often presented on the slides in a relatively static manner, even though the dynamic nature of programming. However, if not done correctly, students can easily be overwhelmed by the amount of code that is presented at once and this gets even worse if the remainder of the slide set build onto this code (Brown & Wilson, 2018). Furthermore, using such a static method to explain a programming concept (that is often dynamic) can lead to some shortcomings in the course: students cannot tinker with the code themselves and the teacher cannot show the students what happens if you change something (Majherová & Králík, 2017). Moreover, the complete coding process, including the reasoning, debugging and alternative solutions, are not covered, even though these are essential to become a good programmer (Rubin, 2013).

Lecture is theoretical, but coding is practical

According to literature, students teach programming mostly by doing, as it is a skill learned by practicing. Contradictory, in the traditional lecture setting, theory about Computing Science is mostly presented and no practice is present at these lectures for the students (Santos, Tedesco, Borba, & Brito, 2020). Often, a learning activity about coding is combined with several lectures and an assignment on which students can work afterwards, but we can identify several shortcomings in the skills of the students which are related to the aforementioned problem. Even though students are familiar with the presented theory, it is still challenging for students to translate that into their assignment, as they have not seen the theory in practice, i.e. a coding environment or their IDE (Rubin, 2013). Furthermore, students are not familiar enough with using the debugger or other helpful tools in the coding environment, as these are not discussed in the lecture (Medeiros et al., 2019) (Gutierrez et al., 2022).

Little interaction

In all lessons, teachers are aiming for interaction in the classroom, such that students ask questions and the teacher can ask questions to the students. This is especially true in a programming course, where there are a lot of difficult and abstract concepts, there are often multiple solutions possible and students often ask themselves "what happens if..." etc (Santos et al., 2020). In practice however, there is often little interaction in the (programming) lecture or none at all. As a result, students have difficulty with focusing in the lecture and are less motivated. Additionally, without interaction, the students are only listening at the teacher and not actively thinking about the material, which has a negative impact on the learning goals (Brown & Wilson, 2018). Furthermore, a lack of interaction results in the teachers' misunderstanding of the actual misconceptions of students. So, there is a possibility that the teacher is elaborating on concepts that are clear to students, but worse, also glancing over topics that are not yet clear (Qian, Hambrusch, Yadav, Gretter, & Li, 2019).

2.1.2 Solutions

Live coding

As previously mentioned, in the traditional lecture setting, code is often presented on slides, which results in students being overwhelmed and the absence of the thought processes and bug fixing which students encounter when actually writing code themselves (Brown & Wilson, 2018). A solution for this would be live coding sessions, in which the teacher writes the code from scratch or a simple skeleton as starting point. Next to the thought process, the students can see the error handling, but also shortcuts and workarounds in the used IDE. Moreover, students that do not completely understand the material can catch up with these lessons, such that everyone is still getting along (Sharma et al., 2020b). The research of Rubin (2013) have shown that students that watched live coding sessions got higher grades, these students became better at recognising and solving common bugs and experienced the live coding sessions as more educational than code presented on slides.

Formative assessment (via an online quiz)

For both teachers and students, it is beneficial to have small exercises in a lecture with which students can practice, which function as formative assessment (feedback for teachers and students about the discussed material) (Santos et al., 2020). The teacher engages the students with these exercises and with the students' answers, the teacher can monitor whether the students understood the topic and which concepts should be elaborated further. For the students, it is helpful to immediately practice with the new material, especially in a programming course where applying the material in practice is one of the key elements of the course (Tillmann et al., 2012). It would be ideal if the students could write a short program themselves in the lecture as an exercise, but that is not always possible. As an alternative, the teacher can create exercises about shorter fragments of code and ask the students what the expected behavior is, or give a desired functionality and ask the students to select the correct code fragment from different code fragments that executes that behavior. Additionally, these exercises can point out the common pitfalls that students often make, such that students are alert for these (Brown & Wilson, 2018). An online quiz is an effective method to carry out the exercises: the teacher gets response from all students, the teacher can immediately monitor how well the students did and the students are more engaged, because they can use their phones which offers variation in the lecture (Tillmann et al., 2012).

Easy examples

As seen, students are easily overwhelmed by the examples shown in the lecture, as these are often too large or too complex, especially if the students do not understand the fundamentals of programming. To tackle this issue, it is beneficial if the teacher actively thinks critically about the structure of the lecture, especially the given examples, which could easily be overlooked (Majherová & Králík, 2017). It is helpful if the teacher starts with easy examples and gradually builds up to more complicated examples, such that the students do not get lost and are not overwhelmed. A strategy that simplifies examples is labelling smaller parts of the program, such that the teacher can easily reference these in later examples (Brown & Wilson, 2018). Furthermore, these labels help the students to see patterns in different programs which helps them to understand the examples and use these patterns in their own programs as well.

Authentic examples and exercises

To motivate students more in a programming course, it helps to have authentic example programs in the lecture and authentic exercises for the students (Sharma et al., 2020b). Authentic tasks are tasks with a clear purpose or context, e.g. programming a small game as an exercise or having a program relevant to a real-world problem. Students are more engaged with these authentic tasks and as a result, students have a better understanding of the covered material.

2.2 Teaching Object-oriented Programming

Even though a lot of research focused on the problems and solutions for teaching and learning programming in a general, there is also research done that focuses on teaching and learning Object-oriented Programming specifically. The aforementioned problems are also occurring in this field, e.g. that there is often no participation in the lectures, that the teacher only speaks in lectures and that the students get little or no practice (Yu, Yang, & Wu, 2021). However, as Object-oriented programming is described as more difficult by students (especially if they have only programmed imperatively before), there are some other, more specific problems that are arising.

These problems and the solutions for learning and teaching Object-oriented programming are discussed below:

2.2.1 Problems

Abstract nature of Object-oriented Programming

Students already struggle with the abstractness of imperative programming (Medeiros et al., 2019), but they struggle even more with the abstract nature of Object-oriented programming. A lot of students describe the concepts of Object-oriented programming as "vague" and non-intuitive (Xinogalos, 2015). This leads to even more challenges when teaching Object-oriented programming, as it is already difficult to teach imperative programming. However, the fact that students describe Object-oriented programming as "vague" has also other reasons, which are discussed further below.

Students focus on functionality of program instead of structure

When students are starting to learn programming, they begin with imperative programming, of which the focus is often the functionality of the program itself. However, when students are taught Object-oriented programming, the focus is not on the functionality anymore, but mostly on the structure of the program. Unfortunately, students are still under the impression that functionality is the most important part of the program and are not focusing on (and also missing the importance of) a well thought-out program structure (Stamouli & Huggard, 2006) (Gutierrez et al., 2022). This can also be seen in the debug process of students in Object-oriented programs: students often solve the bug by adding more code to a specific class (which is not meant to got there) without thinking about the complete structure of the program, which leads to bad coding habits and possibly more bugs (Chibizova, 2018).

Students do not have enough programming experience

The basic Object-oriented programming concepts and structures are for an experienced programmer very intuitive and understandable, but in experience, students that are relatively new to programming do not find these concepts easy to understand at all (Chibizova, 2018). The main issue here is that the students see these concepts early in their programming career, but they simply do not have enough experience yet to understand the purpose of Object-oriented programming. Students did not encounter enough problems yet related to their design of a program and as a result, are not familiar with design analysis and the usage of UML diagrams for example (Gutierrez et al., 2022). On a positive note, research has shown that these problems slowly fade away during a Object-oriented programming course, as they students have seen enough examples and case studies at that point (Sun, Wu, & Liu, 2020).

Fundamental concepts are often unclear for students

Another important problem that is often overseen is that the fundamental concepts of Object-oriented programming are often not clear for students. Examples of these misunderstood concepts are the difference between Classes and Objects, but methods as well, as pointed out by both Xinogalos (2015) and Gutierrez et al. (2022). A possible reason for this could be that students are still having trouble with the idea of modularising code and how to apply this idea effectively in their programs (Chibizova, 2018). Similar to these fundamental concepts, students are struggling with the usage of polymorphism and overloading, which are also essential for understanding and applying Object-oriented programming (Sun et al., 2020). Similarly to programming in general, teachers are often not aware of these misunderstandings, so teachers often have misconceptions in this field as well regarding the understanding of the students (Qian et al., 2019).

2.2.2 Solutions

Clear examples and structure

Similarly to programming in general, it is important that there is a good structure in the slides: starting with simple examples and gradually building up to more complicated examples (Yu et al., 2021). Specifically for Object-oriented programming, this holds for new concepts as well. For example, when looking at design patterns, it is helpful to start with simple patterns (Factory Pattern) and discuss more complex patterns (Visitor Pattern) when students understand the basics of design patterns. Furthermore, as Object-oriented programming focuses mainly on the structure of the program instead of the functionality, it is helpful to actively include UMLs (Unified Modeling Language) with the corresponding programs (Agbo, Oyelere, Suhonen, & Adewumi, 2019). Not only does this visualisation help the students, but by actively seeing UMLs in the lectures, students will use UMLs more often themselves (explicitly or implicitly). Moreover, as it is often not clear for students why certain design choices are better, it can be helpful to introduce new concepts by an example that does not use the new concept, which illustrates its purpose immediately (Stuurman, Passier, & Barendsen, 2016).

Interactive lectures

As Object-oriented programming is a bit different than programming in general, Yu et al. (2021) indicated that it is even more important that the teacher tries to maintain an interactive learning environment. This can by done by asking questions or handing out a preparation assignment for students before the lecture, such that a two-way interaction is created between the teacher and students, instead of the traditional one-way. Moreover, Yu et al.'s observations indicated that it could be beneficial for the teacher to look into problem-based teaching methods, where students learn by immediately solving bigger problems and figuring concepts out themselves instead of having lectures.

Clear criteria in assignments

As discussed before, the students' perception of program correctness of mostly related to the functionality of the program instead of the structure. So, for students it is often not intuitive that the structure of their program plays a large role in the practical usage of the program or their grade (Stamouli & Huggard, 2006). Moreover, the perception of correctness is heavily influenced by the criteria of the assignment they have to make. Therefore, it is important that teachers give a clear criteria for the assignment; not only the functionality they desire, but most importantly, the desired structure that the students should practice with.

"Internet +" teaching

Yu et al. (2021) stated that Object-oriented programming could effectively be combined with "internet +" teaching, which is the integration of the internet in education. One suggestion is to combine blackboard teaching with multimedia: explain topics related to practical subjects through multimedia (e.g. a video or a live demonstration), but explain the more complex topics (which are more difficult to explain through multimedia) on the blackboard (Yu et al., 2021). This combination not only motivates the students by the variety in the lecture, but it improves the teaching efficiency as well, as simpler subjects can be explained relatively quickly through multimedia. Furthermore, it is helpful to provide several online references beside the material in the lectures (Yu et al., 2021). Not only do the students have more support in studying the new material, but they also learn how to effectively use the internet to find the information they need. Lastly, it is useful to have an online platform for students where they can ask questions to encourage peer learning as well.

2.3 Teaching Design Patterns

As discussed, Object-oriented programming is already a challenging topic for students and is often investigated in research. Design patterns are really relevant for Object-oriented programming and are very often present in an

Object-oriented programming course (Azimullah et al., 2020). Similar to Object-oriented programming, students find the design patterns a difficult topic to understand and there are several research papers that investigates why this is the case. Note that even though design patterns are a challenging topic, there is significantly less research done on design patterns compared to Object-oriented programming. The observations in the literature regarding design patterns are discussed below:

Abstract nature of design patterns

Similar to learning (Object-oriented) programming, the main challenge lies in the abstract nature of design patterns for students (Azimullah et al., 2020). This abstraction is related to the grouping of similar functions or attributes to Objects, without knowing the explicit definition (Silva et al., 2019). Especially combined with the findings that the fundamentals of Object-oriented programming are often unclear for student (Gutierrez et al., 2022), the abstractness of design patterns are even more challenging for students to understand.

Unclear when to apply design patterns

Another big problem teachers experience in their classes about design patterns is the fact that students are not able to recognise patterns in a program. The reason for this is that teachers often focus on the implementation of a design patterns, instead of the recognition of design patterns and the cases where to use them, which is perhaps even more important than the former (Lotlikar & Kussmaul, 2022). Moreover, design patterns are often categorised as "a catalogue"; design patterns are explained with the use of one specific design problem in a program and this results in students that find it difficult to determine whether a pattern should be applied (Stuurman et al., 2016). Therefore in practice, students want to apply as many patterns as possible in a program which contradicts with the pursuit that programs should be as simple as possible, but also results in students that apply incorrect patterns in programs, thus only making the program more complicated or incorrect (Silva et al., 2019).

Purpose design patterns not clear

Related to problems with Object-oriented programming, students do not have enough programming experience to understand the utility and benefits of design patterns (Mello Fonseca et al., 2019). They simply have not seen enough programs and software designs to see why indeed design patterns are very useful to apply (Stuurman et al., 2016) and how a correctly applied pattern helps in producing software in larger groups (Lotlikar & Kussmaul, 2022). As a result when writing Object-oriented programs, students jump straight into the implementation of the described problem instead of thinking about the program structure beforehand, which is one of the key elements of Object-oriented programming (Lotlikar & Kussmaul, 2022).

Static explanations for dynamic concepts

One of the most important elements of design patterns are their dynamic nature; the manner in which the different functions and classes are related to each other and how they are interacting. However, in the current lectures or books, these design patterns are explained in a static manner; the UML of the design pattern is shown, but the actual interactions between the classes and functions are not clearly shown. So, explaining these design patterns in a dynamic environment (e.g. an environment where students can actually experience and see the interactions) would really benefit the students' understanding of design patterns (Azimullah et al., 2020).

Solutions

One tool to tackle the aforementioned problems is to use UMLs when explaining design patterns, which proves to be very helpful in practice. However, using UMLs also comes with some drawbacks, e.g. UMLs are still a static representation used for the dynamic design patterns and UMLs can quickly become very big and complex, especially working with larger design patterns (Azimullah et al., 2020). Furthermore, studies found that not all students are confident with reading and creating UMLs, which makes the representation of design patterns in UMLs only more confusing (Azimullah et al., 2020).

Lotlikar and Kussmaul (2022) gave some guidelines in their paper on how to teach design patterns, for example:

- First focus on the context, problem and consequences, then on the implementation
- Students should give a rationale for all the design patterns they use
- Let the students discover a well-known design pattern by themselves.

Unfortunately, we could not find any research that provided a concrete plan for teaching design patterns to solve the discussed problems yet. Therefore, in our research we combine the previously mentioned problems to design a structured learning activity to teach design patterns and evaluate it afterwards. To effectively evaluate our activity, we inspected how other literature tackled such an evaluation.

2.4 Evaluating a learning activity

A lot of research related to programming education are designing a learning activity or a learning tool and have to evaluate it or inspect the students' understanding of the new material. To see which methods are effective for these types of research, we looked at the most occurring and effective methods. Note that the first three methods are related to collecting data and the latter two points are related to analysing the collected data.

Asking the students' opinion

To gather data, most research asked the students' opinion of the learning activity one way or another. The most occurring method is the use of a questionnaire or survey that is send to all the participating students (e.g. Xinogalos (2015), Lotlikar and Kussmaul (2022) or Keung, Xiao, Mi, and Lee (2018)). Another method of collecting data is the usage of semi-structured interviews with the students (e.g. Stamouli and Huggard (2006) or Lytle et al. (2019)); the advantage of these interviews is that the researcher can get more information of the students, but the disadvantage is the time needed to conduct and transcribe these interviews.

Inspecting students' solutions

Another method that we saw a lot to collect data was the inspection and analysis of the students' work where they had to apply the material covered in the learning activity. This data can be the solutions of the students of an assignment or exercise (e.g. Hayashi, Fukamachi, and Komatsugawa (2015) or Eppley and Dudley-Marling (2018)), but could also be the exam results of students (e.g. Xinogalos (2015) or Anggrawan, Ibrahim, Suyitno, and Satria (2018)).

Observe the students

A method that we saw less, but was still used by a few researchers, was the observations of students without the opportunity to ask them questions or interact with them. These were either done via screen recordings of students while they were working on the exercises (Stuurman et al., 2016) or by literally observing the students while they were working in a practical session (Lytle et al., 2019).

Triangulation

One element that we saw in a lot of the research papers is the triangulation of different data sources, to strengthen the presented results in the research. In all the instances where the researchers applied triangulation in the analysis, the researchers had access to solutions of students which were analysed and used at one point questionnaires, interviews or recordings of students to get their opinion. With these combinations, the researchers had access to the final solutions of the students, but could still get their opinions or their thought processes when making these assignments. Examples of research that applied triangulation are for example Hayashi et al. (2015) (looked solutions to an exercise, exam and had send a questionnaire to the students), Lytle et al. (2019) (looked at solutions to the exercise and had semi-structured interviews with students) and Stuurman et al. (2016) (looked at solutions to the exercise and at the recordings of the students while working on the exercise).

Experimental and control group

A method that would come to mind to analyse the gathered data related to the learning activity would be the usage of a experimental and control group, with which the researchers can clearly compare results and see differences. However, this method is not applied that often, probably because using a experimental and control group in an educational setting is often conflicted with the ethical and practical aspects of the educational setting. Nonetheless, examples of research papers that applied this method are Khakim (2019/08) and Lotlikar and Kussmaul (2022).

Chapter 3

Goal

Research Question

How can a learning activity contribute to the students' understanding of design patterns?

Research Subquestions

- **RSQ1:** How can the students' application and knowledge of design patterns be characterised after the learning activity?
- **RSQ2:** How did the students perceive the designed lecture?

Chapter 4

Setting

This research was conducted at the Radboud University at Nijmegen. First year Computing Science students and Artificial Intelligence students were following a course Object-oriented programming here, which consisted of fourteen weekly lectures, where the basic concepts of Object-oriented programming were explained and applied by the students. The students did a programming course ”Iterative Programming” beforehand, which covered the basics of programming, for example loops, recursion and basic sorting algorithms. In the year of this research, around 300 students enrolled for this course, but it was unclear how many students were still actively participating when the design patterns were discussed.

Every week, a physical lecture is given by the professor, which introduces the new concept in a traditional lecture setting (students are mostly passive listeners while the professor talks). In addition, this lecture was also a livestream, such that students could directly view the lecture from home and were able to ask questions via the chat. Afterwards, the used slide set and the recording of the lecture were shared with the students via the online learning platform.

This lecture is followed by a physical tutorial session, in which the newly taught concepts are presented from a more practical perspective with several examples of code. This tutorial session serves as a bridge between the theoretical lecture and the practical sessions that follow. Similar to the lecture, the tutorial session was a livestream and the presented slides and the recording of the session were shared with the students afterwards.

After the lecture and the tutorial session, the students had the opportunity to participate in the practical sessions. The physical practical sessions were sessions of 4 hours in which the students could work on the programming assignment of that week to apply and practice with the new material. These assignments are often made in pairs and several teaching assistants were present at the practical sessions to immediately help students if they get stuck.

These are the three weekly moments of contact of the course Object-oriented programming with which the teachers provide enough contact hours to support the students in the course. Unfortunately in practice, there are only a few students that are present at the lecture and there are even less students present at the tutorial sessions. Luckily, there are a lot of students present at the practical sessions.

To test the students' knowledge of the topics covered in the course, there is an exam at the end of the course. This is a digital exam, but students do not have access to a compiler or IDE when making this exam.

There is one lecture dedicated to design patterns in this course (week 10), so that is the week in which we conducted our research. In the sections below, the lecture, tutorial session, assignment and the exam are discussed in more detail. For our research, we made changes to the lecture and the assignment; the tutorial session was outside the scope of this research and we used the students' solutions for the exam as a data source.

4.1 Lecture

The lecture was the main moment of contact we wanted to investigate, as this lecture was already given in a setting in which many students could participate (students could be present live, watch the livestream and or the recording afterwards). We could adapt the material that was taught in the week dedicated to design patterns, but the exact changes had to be discussed with the professors to see what was possible, as it had to be in line with the curriculum.

4.1.1 Original lecture

The focus of the original lecture dedicated to design patterns was mostly discussing different design patterns and presenting one specific implementation of each pattern. The discussed design patterns were:

- Singleton Pattern (Explanation)
- Strategy Pattern (Explanation)
- Decorator Pattern (Explanation)
- Optional Datatype (Explanation)
- Visitor Pattern (Explanation)

4.1.2 Changes & Approach

Firstly, we evaluated the learning goals of the week dedicated to design patterns with the professors, as the original slides were mainly focused on the implementation of design patterns instead of the application and usage. However, we concluded that implementing a pattern is still a relevant learning goal, but actually applying and recognising patterns is another, perhaps even more important, learning goal. So, our learning goals for the designed lecture were:

1. Students can implement the discussed design patterns
2. Students can apply and recognise the discussed design patterns

To accomplish both learning goals, we decided that we reduce the number of discussed design patterns, but instead we discuss the remaining patterns in more detail as well as the application of these patterns.

These are the patterns which were presented in the lecture:

- Strategy Pattern
  This is a relatively easy pattern and students had already used variants of this pattern previously in the course (without the knowledge that it is a design pattern). Therefore, this pattern served as a decent pattern to start with.

- Decorator Pattern
  This pattern is relatively easy to understand for students, because it has some clear real-world applications. Additionally, this pattern is a structural pattern (a pattern to construct effective datatypes and objects) instead of a behavioral pattern (patterns that focus more on algorithms and how objects interact with each other), so presenting this different type of design pattern was useful.

- Visitor Pattern
  This pattern is a more complex pattern, but a lot of interesting things are used in this pattern regarding typing and interactions between objects. Furthermore, this pattern was used in the original assignment, so it was sensible to keep this pattern in the lecture, so we could keep the original assignment as a starting point.

In the literature, we saw several common problems related to teaching design patterns, using UMLs and teaching Object-oriented programming (OOP) in general. In the changed lecture, we aimed to tackle some of these problems as well (which are discussed in chapter 2):

- Students focus on the functionality of the program instead of its structure (OOP)
  To tackle this, we aimed in this lecture for an explicit focus on the structure of the program, instead of the functionality of the program. The approach we aimed for was to state a problematic, but easy-to-understand program, followed by a discussion of the possible structures

of the program and discussing why these are effective or problematic. Furthermore, we started the lecture with some introductory exercises related to structures of programs instead of its functionality, to show the students that the structure will be the focus of the lecture.

- The teachers do not have the correct understanding of the students' misconceptions (OOP)
  Especially with students that are new to programming, teachers are not always up-to-date with the students' misunderstandings. To tackle this issue, we wanted to use an online quiz with exercises, such that we can get input of most students and see which concepts were not completely clear.

- Students find design patterns too abstract (design patterns)
  This problem can be tackled by introducing the design pattern using a concrete use case, but the disadvantage of this is that the pattern will "overfit" this use case. Therefore, we still present the pattern with the help of a UML and one concrete use case, which both will be discussed in detail, but additionally, we provide enough other examples of programs in which the pattern can be applied or definitely not.

- Students find it difficult to see when to use patterns (and when not) (design patterns)
  This problem is immensely related to our observation of the learning goals; there should be more focus on the application of design patterns. A solution would be to present many examples of cases where the pattern could be applied and problems were the pattern is not usable. Our idea was to provide these examples in the form of exercises in the online quiz, to not only show these examples, but let the students actively think about them as well. Moreover, if we have some examples where, at first glance, the design pattern can be applied, but this is actually not the case, we stimulate the critical analysis of the students as well.

- Static explanations are used for the dynamic behavior of design patterns (design patterns)
  We hoped to tackle this problem by adding a live coding session in which we presented the pattern implemented in a functioning code project. The debugger can be used here to show the flow of a program and the interactions within the design pattern, thus making the dynamic functionality of the design pattern more visible.

- UMLs in lectures are complex and too big to properly understand (UML)
  The UMLs used to present the design patterns are often huge and and complex, but it is impossible to make these UMLs smaller and

easier. Therefore, we break down every UML presented in the lecture in smaller parts, such that the whole UML can be understood more easily.

- Not all students are confident enough with reading UMLs
  Again, by breaking the UML apart in smaller bits tackles this problem. Furthermore, as introductory exercises, we ask students several questions related to smaller UMLs, such that the basics are understood by all students and that they are all on the same page.

Moreover, there are some elements that a teacher can put in a programming lecture (or a lecture in general), which makes the lecture easier to follow and more interactive, according to the literature (See chapter 2 for more details). This mainly has to do with the way to structure the lecture (or parts of the lecture):

1. Introduction
   Start with an introduction to get the interest of students. For example, show why this new material is useful or some introductory exercises.

2. Explanation of new material
   In this phase, the new material is explained, preferably in a clear step-by-step manner.

3. Live coding session
   Next to presenting code on slides, it is beneficial to show the concept in a coding environment as well. This serves as a helpful bridge between the theory and practice, as the students have to apply the new concepts themselves in such a coding environment. Furthermore, a change of "environment" in the lecture helps the students to concentrate, as it serves as a change of pace.

4. Exercises in an online quiz
   To let the students participate in the lecture and let them actively think about the material as soon as possible, it is useful to have some small exercises related to the previously discussed concepts. Using an online quiz, students not only have to use their phones, which is another change of pace, which contributes to the focus of students, but more importantly, the teacher receives the answers of (almost) all students, so the teacher gets an insight whether the students understood the material.

5. Conclusion
   Wrap everything up in a conclusion and move on to the next topic or finish the lecture.

With these things in mind, we designed a lecture that gave at the course Object-oriented programming in the week dedicated to design patterns.

4.1.3 New lecture

The slides for the designed lecture can be found in appendix A. Note that every bullet point on the slides appear one by one, so that the students do not get overwhelmed if they saw the complete slide. In the course of the lecture, students could participate in an online quiz, of which the questions and answers can be found in the slides, but also separately in appendix B.

The outline of the slides was as follows:

- Introduction (slides 1-13)
  Here we introduced the topic of design patterns to students, mostly via several exercises in the online quiz (questions 1-4). The first two questions were simple exercises about the relation of programs and UMLs, to make sure that all the students were on the same page regarding understanding UMLs. The other two questions were about comparing programs with each other and whether there was overlap in their structure, to serve as a starting point for understanding the concept of design patterns.

- Strategy Pattern (slides 14-34)
  How the design patterns were exactly explained, will be discussed later. For the Strategy Pattern, the questions 5-8 were used.

- Decorator Pattern (slides 35-57)
  The questions 9-12 were used in this section.

- Visitor Pattern (58-80)
  The questions 13-15 were used in this section.

- Recognising patterns (slides 81-91)
  The purpose of this section was to explain to the students why we have these design patterns; mainly that we do not expect that students know these patterns by heart, but that they recognise similar patterns in programs and reuse effective solutions. Moreover, we made comments on how these slides and the assignment are related to each other, to create a bridge between theory in the lecture and practice in the assignment. We also added several question in the online quiz that dealt with recognising a pattern out of multiple patterns instead of stating whether a specific pattern could be applied (questions 16-18).

- References (92-94)
  Here we added references to students with a clear instruction what students can find on these websites, such that they can easily see which are useful for them. Furthermore, we added a link to the questionnaire, which will be discussed in chapter 5.

To explain the different patterns, we used the following structure, which was based on the literature findings of the previous section:

1. Stating a problem which can be solved by that pattern
   As an introduction, we present a students a simple, concrete program which leads to some implementation problems if we implement it in a naive manner. From these problems, we suggest a new, concrete implementation which solves this problem (presented with a UML), which is actually an application of the discussed pattern. This program will be the main example provided for the discussed design pattern.

2. Generalising the previous solution into the design pattern
   From this concrete application of the pattern in the previous program, we generalise this approach into the abstract design pattern, presenting the UML.

3. Discussing the UML of the abstract design pattern
   To make sure that the students are not overwhelmed by the UML, we discuss all its separate parts in detail. This ensures that the students understand all the smaller parts and thus, the complete UML becomes clear as well.

4. Discuss the UML of the concrete implementation of the previous program
   Now that the abstract design pattern is (hopefully) clear to the students, we can discuss the UML of a concrete application, which is the program for the introduction. Note that the students already have seen this UML (without knowing that this pattern was applied), so with the repetition of this UML and the new understanding of the design pattern, students can more easily understand this application.

5. Live coding session
   Students have seen the abstract UML of the design pattern, as well as the UML of a concrete application, but students have not seen the code of a concrete implementation. Therefore, we wrote out the code for the presented program and showed it to the students in a live coding session. Not only did the live coding session made the design pattern more concrete, but these code projects were also provided to the students afterwards, which could help students with making the assignment.

6. Exercises in the online quiz
   After all these explanations, students could immediately apply the newly learned knowledge in a few exercises in the online quiz. Most design patterns started with a question in which the students had to indicate the correct UML for the design pattern, to see whether they

had a high-level understanding the design pattern. Afterwards, the questions were about whether the discussed pattern could be applied in a specific program. These questions served as more examples of the application of the pattern, but as well as practice for students to apply the patterns, which is often skipped in lectures of design patterns. All the questions, answers and elaborations were provided in the slides.

7. Recap
   At the end, we had a slide that recapped the discussed design pattern.

For the design patterns, we used the following example programs:

- Strategy Pattern
  We presented a navigation app that finds a route from x to y. However, there were different options to find such a route, e.g. a route for cars, for bicycles or via specific buildings, similar to Google Maps for example. These different options served as clear examples of strategies, thus this was a practical example of an application of the Strategy Pattern. The code for the live coding session can be found in appendix C.1.

- Decorator Pattern
  Here, we presented a program that could notify the user. However, the problem was that there were multiple platforms on which users could be be notified and the user could specify on which platforms he wanted to be notified. We first presented a solution where every combination was implemented as a separate object, but this was obviously not a practical solution, so the proposed solution made use of the Decorator Pattern. The code for the live coding session can be found in appendix C.2.

- Visitor Pattern
  Here we used the example of a shop that sold different items, but multiple actions could be performed on (a list of) these items. We start with a proposed implementation of the Strategy Pattern here, but we ran into some problems here (using `getClass()` etc). We solved this problem, but new problems arise. Therefore, we keep proposing solutions until we had a solution which looked good, which was the Visitor Pattern. We made a comment that this pattern looked very complicated and elaborated why we still want to use it. The code for the live coding session can be found in appendix C.3.

4.2 Tutorial

The tutorial session was not in the scope of this research. In the given tutorial session, basic code for the assignment was provided to the students and some hints on how to proceed. Note that there were only a few students that actually went to the tutorial session.

4.3 Assignment

After the lecture and the tutorial session, students had to make an assignment. The assignment about design patterns consisted of an implementation of the Visitor Pattern, which was perceived as difficult in earlier experience. For the new assignment, satisfying both learning goals of this week, we decided to split the assignment in two parts: part 1 was related to recognising a pattern and refactoring the program and part 2 was an implementation of a specific design pattern. The PDF of the updated assignment can be found in appendix D. Both parts are discussed below:

Part 1

The students received a small program that could encrypt strings with relatively simple encryption methods: Caesar3 (explanation), ROT13 (explanation) and MIRROR, which simply spells the string backwards. The students were asked to refactor this program with one of the discussed patterns, which was supposed to be the Strategy Pattern.

The provided code can be found in appendix E.1, which used a simple enumeration for the different encryption methods. When applying the Strategy Pattern, students will write a program similar to the solution shown in appendix E.2. However, as Caesar3 and ROT13 could be generalised into an encryption method that simply "shifts" the letters inside the string with a certain number of places, students could generalise this program even further. Even though we do not expect that students see this solution, we still provided this solution, because it is still a "better" solution. The code of this solution can be found in appendix E.3.

We deliberately kept this program and the first part of the assignment relatively small, as the second part was expected to be a lot more difficult and larger.

Part 2

Part 2 was mostly the original exercise, but we simplified it a bit, mainly by providing more support for the generic Visitors (students had seen generics before, but we expected that they still had difficulty with it). In essence, the students had to implement the Visitor Pattern. In this exercise, the students had to implement boolean formulas (True, False, variables and the basic operators `not`, `and`, `or` and `implies`). Note that they were tasked to implement a binary operator for `and`, `or` and `implies` using the Strategy

Pattern , a Standard Library and an enumeration. The students had implemented a similar formula structure earlier in an assignment. Afterwards, the students had to make two Visitors: a visitor that evaluates the given formula to its boolean value (EvaluateVisitor) and a visitor that shows the presented formula with the minimal number of parentheses (ShowVisitor). Before the implementation, the students had to create a UML of the program and afterwards, the students had to make a sequence diagram of the function calls in the Visitor pattern.

There was minimal code provided, which can be found in appendix F.1 and a solution in appendix F.2.

This assignment completes the week dedicated to design patterns, so the designed learning activity.

4.4 Exam

At the end of the semester, the students had to make an exam, which contained an exercise related to design patterns. This design of the exam was also outside the scope of the research, but we used the students' submissions as a data source. The exam question and its answer can be found in appendix G.

Chapter 5

Methodology

In this chapter we discuss our used methods to gather the data needed for our research.

To answer **RSQ1**, "How can the students' application and knowledge of design patterns be characterised after the learning activity?", we need to look at the students' programs to see whether they understood the concepts, but also the process of how the students got to the final program, such that we can see which problems they encountered. Moreover, it is useful to ask the students how the experienced they assignment, to see if we see the same results or if we see other observations. For **RSQ2**, "How did the students perceive the designed lecture?", we need to ask the students how they experienced the lecture.

In total, we had six moments to collect all the necessary data from the students to answer all the research subquestions:

1. Online Quiz
   We analysed the students' answers to the online quiz in the lecture to see if there were any clear learning effects.

2. Questionnaire after lecture
   The students were asked to fill in a questionnaire immediately after the lecture with which we got insight in how the students experienced the lecture.

3. Interviews in the practical sessions
   We interviewed several groups while they were working on the assignment in the practical sessions. Here we collected data about the problems students were facing in the assignment, but also feedback on the lecture and the tutorial session.

4. Screen recordings of the student groups
   We asked a small number of groups if they could record their screen

while they were working on the assignment. With these screen recordings, we could get detailed information of the students' coding process which was not visible from the other data sources.

5. Questionnaire after the assignment
   All the groups had to fill in an obligatory questionnaire after finishing the assignment, in which we asked for the problems students faced and how they solved these problems. With this questionnaire, we got high-level descriptions of the problems students were facing, but from all the groups.

6. Exam results
   At the end of the course, we analysed the students' submissions of the exam. With this data source, we got to analyse actual code from the students to see which concepts the students did understand.

To combine all these different data sources, we triangulate this data at the end to see which things we see in different data collections.

All the different data sources are discussed in more detail in the sections below and how they contribute to the research questions.

5.1 Online Quiz

From the online quiz that we used in the given lecture, we took a look at the given answers to see any learning effects to support the answer for **RSQ1**. We exported these answers to an Excel sheet of (anonymous) students and their given answers.

To analyse these results, we started by calculating the number of correct answers and its percentages for all questions, and for all students, the number of correct answers and its percentages. For these percentages, we took into account the number of answered questions.

We also categorised the questions in the subjects (Introduction, Strategy pattern, Decorator pattern, Visitor pattern and Pattern general) and see what the percentages of correctly given answers were for these categories to compare the different patterns. Additionally, we looked at how many students actually gave an answer to these categories, to see if the number of answers in the course of the lecture.

Furthermore, we categorised the questions in the following categories:

- Introductory (I) (questions 1, 2, 3 & 4), for introductory exercises
- UML (questions 5 & 9), for exercises about UMLs
- Application Easy (AE) (questions 6, 7, 11, 13 & 15), for easier exercises in which the students had to answer whether and how a certain pattern could be applied

- Application Difficult (AD) (questions 8, 10, 12, 14), similar to AE but for more difficult questions

- Pattern General (PG) (questions 16, 17 & 18), for questions were the students had to state whether one of the presented patterns could be applied.

For these categories, we calculated the correct answers per category and also inspected the individual percentages to see if something interestingly could be observed in the course of the lecture.

After calculating all these values, we inspect these to see if there were any interesting observations related to the learning effects to answer **RSQ1**.

5.2 Questionnaire after lecture

To get some feedback on the lecture of the students, we asked the students to fill in a (non-obligatory) questionnaire after attending or watching the lecture. These results were then used to mainly get data for **RSQ2**, but also for **RSQ1**. The questionnaire can be found in appendix H, but the students had to fill it in online. Note that the questions with ovals in front of the answers are multiple choice questions and exactly one question had to be selected.

More specifically, we aimed to get feedback on the following points (with the added questions to receive this feedback):

- Interesting statistics:
  - What are you studying? (multiple choice)
  - How did you follow this lecture? (multiple choice)
  - Could you focus this lecture? (multiple choice)

- How the students perceived the lecture and specific lecture elements:
  - How did you experience the explanations of design patterns?
  - How did you experience the used UMLs in the slides?
  - How did you experience the exercises in the lecture?
  - How did you experience the programs presented in Netbeans instead of code snippets?

- A comparison between this lecture and their other lectures:
  - How did you experience this lecture compared to the other (Object-oriented programming) lectures?

All the answers of the students were coded and analysed, where we coded every question separately. As we had only a vague idea of the codes that would occur, we would start by coding what we see, starting with specific codes. Afterwards, when we had a better view of the overall codes, we could combine codes and generalise observations. With the codes questionnaire results, we could see the general comments of the students and their overall experience of the lecture. We could compare the answers of all the different questions to each other, but most questions were quite distinctive. Therefore, we looked at every individual question and what the most interesting and consistent observations were. For each question, we discuss why it was added to the questionnaire and which codes we expect to apply (while analysing the answers, there will be codes added which we did not think of beforehand).

5.3 Interviews at the practical sessions

Three days after the lecture, students had to work on the assignment. Many students went to the two practical sessions to get support of several teaching assistants (TAs) to work on this assignment. In these two practical sessions, we were present to walk around and conducted short semi-structured interviews in which we asked how the assignment was going and which problems students encountered. The process we describe below was used for both practical sessions, where the first session mainly consisted of Computing Science Students and the second session of Artificial Intelligence students.

A practical session consisted of 4 hours. So we decided to join the practical session after 1,5 hours, as then most students probably started and even finished the first part of the assignment and perhaps started the second part. We walked through the room and talked to all the groups that were working on the assignment and we asked them several questions regarding the following topics (dependent of the answers, we could ask other follow-up questions):

- Part 1 of the assignment
    - Did you finish the first part of the assignment?
    - Which problems did you encounter?
    - How did you solve these problems?
    - How did you perceive the difficulty of this part?
    - ...
- Part 2 of the assignment
    - Did you start the second part of the assignment?
    - Are there any problems you encountered already?

  - How did you solve these problems?
  - Is the Visitor pattern clear?
  - ...

- Lecture
  - How did you experience the lecture?
  - Any feedback on the lecture?
  - ...

- Tutorial session
  - How did you experience the tutorial session?
  - What would be useful additions to the tutorial session?
  - ...

In addition to the groups of students that we observed, we also got some feedback of several TAs; a few of which were TA for the first time, but some were also TA for a few years, so these provided also interesting insights in the assignment.

The questions about part 1 and part 2 of the assignment are included to get some data for **RSQ1**. The feedback from the students on the lecture is used to answer **RSQ2**. Even though the tutorial session is outside of the scope of this research, it was still interesting to see how the students experienced it. Additionally, their comments about useful additions were used in the discussion for a potential next iteration of the learning activity.

While the students were answering the questions, we did not literally type out their answers, but we simply typed down the keywords of their answers. The reason for this was that we wanted to keep the flow of the conversations and many of the answers could be easily categorised. So, after the two practical sessions, we had two documents with the observations from the practical analysis, which we could analyse afterwards.

The typed out answers were coded afterwards and categorised into the categories Part 1, Part 2, Lecture, Tutorial session and General Remarks. We made some codes beforehand, which can be found in appendix J. If other codes were seen in the analysis, we added these codes, or made the already existing codes more specific. Afterwards, we looked at the coded answers and see whether there were any interesting observations. We looked at every topic individually as they were quite distinctive.

5.4 Screen recordings of the student groups

At the practical session, we asked 5 groups if they could record their screens while they were working on their assignment. With these recordings, we hoped to get more detailed insight into the problems students were facing and how they solved these problems, while the other data sources gave a more high-level overview. This provided data to answer **RSQ1** We asked the students if they could record their voices as well as their screen, but if they only handed in their screen recordings, we were also satisfied.

After we received the recordings, we watched them and saved the time stamps when the students did something worth noting. These transcriptions were coded to see whether there were any interesting observations. Again, we divided the codes into part 1 and part 2 of the assignment and some general remarks. These are some codes we expect to see:

- Part 1 of the assignment
  - Unsure which pattern should be applied
  - New classes were made for the pattern
  - Slides were used to solve problems
  - Code projects were used to solve problems
  - ...
- Part 2 of the assignment
  - Problems with constructing the UML
  - Problems with implementing the Visitables
  - Problems with implementing the Visitors
  - Problems with using the Visitor Pattern
  - Slides were used to solve problems
  - Code projects were used to solve problems
  - ...
- General remarks
  - ...

If there were any observations which we did not expect beforehand, we created extra codes if needed or specify existing codes. Afterwards, we looked at the interesting observations for part 1, part 2 and other general observations.

5.5 Questionnaire after the assignment

With the interviews in the practical sessions, we gathered data from many different students while they were working on it and we saw the students face-to-face. With the screen recordings, we gathered a lot of detailed data, but of a few students. As an addition, we wanted to get some detailed input from students on the problems they faced while making the assignment, which was done via a mandatory questionnaire that students had to fill in after the assignment. This questionnaire focused on the problems students were facing in the assignment, how they tackled these problems and how the slides of the lecture and the assignment were connected with each other. With this questionnaire, we got some insight in the problems the students faced while working on design patterns (**RSQ1**), but also how the students experienced the complete learning activity (**RSQ2**). The questionnaire can be found in appendix I, but the students had to fill it in online. Note that the questions with ovals in front of the answers are multiple choice questions.

More specifically, we aimed to get input on the following points (with the added questions to receive this feedback):

- Interesting statistics:
  - What are you studying (multiple choice)
  - How did you experience the difficulty of the assignment?
- Problems encountered in Part 1 of the assignment
  - Which problems did you encounter in the first part of the assignment?
- Problems encountered in Part 2 of the assignment
  - Which problems did you encounter in the second part of the assignment while constructing the UML?
  - Which problems did you encounter in the second part of the assignment while implementing the formulas (the visitables)?
  - Which problems did you encounter in the second part of the assignment while implementing the visitors (pretty printer & evaluator)?
- The relation between the slides and assignment
  - Were the slides useful for this assignment?

For all questions, we coded the data and analysed these codes to see interesting observations. Similar to the questionnaire after the lecture, we had a vague idea of the codes that would occur, so we would start with specific codes. Once we got a better overview of the occurring codes, we could group and generalise these codes. We made this questionnaire mandatory for the students, so we expect a lot of submissions. Therefore, it was also possible to look at the number of occurrences of codes. After the coding, we looked if there were any interesting observations and we could discuss these results for every individual question, as the questions were quite distinctive.

5.6 Exam results

With the data collected from the interviews, screen recordings and questionnaires, we got a decent insight which problems the students still faced when working with design patterns to answer **RSQ1**. However, it would also be useful to actually look at the programs that the students had written. Even though we could not get insight into the thought process of the students or their coding process, we could still get some information from their final code files. Therefore, we decided to take a look at the students' submission of the Design Pattern exercise in the exam results. We decided to look at the exams rather than the assignment, because at the exam, the students could not use any references to slides, so we could actually see what they picked up from the lecture. Furthermore, in contrast to the assignment, the code written by students in the exam did not have to compile and there was no compiler or IDE available to them, so we could see more clearly which concepts the students did not yet understand. Moreover, we could see the individual students' understanding of design patterns (the assignment was made in groups) and the exam was more practical to analyse than the assignment.

We received the (anonymous) exam submissions of the students and we analysed them by coding the solutions of the students. As the exam exercise consisted of 5 smaller exercises, we analysed each of the sub-exercises individually. As these sub-exercises were relatively small and there was often only one correct answer, we could group each answer in different categories, which we could identify for the most part beforehand. Note that we only looked at mistakes and problems related to the Visitor Pattern they had to implement and not other mistakes that were unrelated to the design pattern. Afterwards, we looked at the interesting observations we could see to help us answer **RSQ1**.

The codes we used for the exam results can be found in the rubric in appendix K. Note that we specify the codes "Visitor call incorrect", "Visit function's body is incorrect" and "Function calls are incorrect" in more detail, depending on the solutions we find in the submissions.

5.7 Triangulation

We collected a lot of data from different sources, so we could triangulate observations which we have seen in different data sources. The idea was that we looked at the results of the different sources and see if there were any results that were seen multiple times. Below, we categorised the possible results, which data sources could support these results and for which research subquestions these results can be used:

Feedback on the lecture

These results were used to answer **RSQ1** and **RSQ2** and the following data sources were used:

- Online Quiz
- Questionnaire after lecture
- Interviews in the practical sessions
- Questionnaire after the assignment

Feedback on the assignment

These results were used to answer **RSQ2** and the following data sources were used:

- Interviews in the practical sessions
- Screen recordings of the student groups
- Questionnaire after the assignment

Students' understanding of design patterns

These results were used to answer **RSQ1** and the following data sources were used:

- Interviews at the practical sessions
- Screen recordings of the student groups
- Questionnaire after the assignment
- Exam results

Chapter 6

Results

In this section, we discuss the results found with the methods discussed in chapter 5. At the end of every data source, there is a summary with the most important observations.

6.1 Online Quiz

55 students participated within the online quiz in the lecture. Due to time constraints, we could not do the last three exercises of the online quiz, so these are excluded from this analysis. We decided to keep all the students in the data set that answered at least one question, as many students missed at least one question.

The average student answered 73% of the answers correctly and the average question was answered correctly by 75% of the students. We can see that the answers were of the correct difficulty, as they were not too difficult, but also not too easy that none of the students gave the correct answer. There are a few outliers:

- Question 6 (100% correct)
  The students were asked whether the Strategy Pattern could be applied in an example program. This was a relatively easy example and there were 2 correct answers (of the 3 options), so that explains why everyone gave the correct answer.

- Question 8 (18% correct)
  This is by far the lowest score of all questions. Again, the students were asked whether the Strategy Pattern could be applied in a given program, but this was a tricky question. At first glance, it looked like it could be applied, but once you thought critically about the program, the Strategy Pattern could not be applied at all. Therefore, many students gave the incorrect answer, but this questions did serve

as a reminder for students to think critically about the application of design patterns, which we can also see in the later questions.

- Question 11 (100% correct)
  The students saw an example program and were asked if the Decorator Pattern could be applied. The application was very clear, so that is the reason why so many students got it correct.

It is also interesting to look at the individual subjects discussed in the lecture. Introduction questions are the first four questions with which we started the lecture. Statistics are represented in table 6.1

Subject	Percentage correct answers	Average number of answers
Introduction	81%	37,5
Strategy	66%	38,5
Decorator	78%	34,5
Visitor	75%	27,3

Table 6.1: Statistics per subject discussed in the lecture

As the introduction contained relatively easy questions, it was to be expected that many students answered them correctly. More interesting is that the Strategy Pattern has the lowest percentage of correct answers, while it is the easiest pattern that was discussed. The reason for this is that it included a question about the UML of the strategy pattern and the previously discussed question 8, which were questions the students found difficult (58% and 18% respectively). However, these questions functioned as a learning moment for students to think critically and pay more attention, as these questions were answered better later in the lecture in the other subjects. Therefore, even though the Decorator Pattern and Visitor Pattern are more difficult, the correctly answered questions are pretty much constant after the Strategy Pattern.

Looking the the number of given answers, we can see that several students joined later in the lecture, as more students participated in the Strategy Pattern than the introduction. Interestingly, not so many students left in the break, as only a few students did not participate in the Decorator Pattern questions, which were after the break. We do see that at the end of the lecture (the Visitor Pattern), a lot more students did not participate anymore with the exercises.

We also separated the exercises into different categories: Introductory (I) for the introductory exercises, UML for the questions about UMLs, Application Easy (AE) and Application Difficult (AD) for the questions with an example program and the question whether the pattern should be applied. The latter category contained easier questions and more tricky questions, so we separated these as well in two different categories. In table 6.1, the percentages of the correct answers can be found and the progression of the correctly answered questions in that category in the course of the lecture.

Category	Percentage correct answers	Progression of the percentages
Introductory (I)	81%	86%, 86%, 79%, 73%
UML (UML)	66%	58%, 74%
Application Easy (AE)	91%	100%, 88%, 100%, 86%, 81%
Application Difficult (AD)	54%	18%, 54%, 85%, 59%

Table 6.2: Statistics per question type discussed in the lecture

We can see that the students struggled with the questions about the UMLs, but in the course of the lecture, the students were getting better in recognising the UMLs, even though the UMLs were getting more complicated. Furthermore, there is a clear difference between the easy and difficult application questions. However, we can still the a similar trend in the difficult application questions as in the UMLs; in the course of the lecture, students were getting better at these exercises, even with the more challenging patterns. These indicate that the students were indeed learning more about these subjects, as the exercises were answered better in the course of the lecture. The introductory questions and the easy application exercises were answered consistently pretty well.

To conclude, the questions in the lecture were of the right difficulty and the students understood how to apply the discussed patterns. Furthermore, we see that in the course of the lecture the students became more familiar with reading UMLs and started to think more critically about in which situations certain patterns could be applied.

6.2 Questionnaire after lecture

The questionnaire after the lecture was filled in by 39 students. With the first 2 questions we could differentiate between the Computing Science (CS), Artificial Intelligence (AI) and Other students and between the students that were attending live, were following the livestream or were watching the recording afterwards. The distribution can be found in table 6.2.

In total, there were slightly more CS students than AI students and of all students that filled in the questionnaire, around 60% were present live, 20% were following live via a livestream and 20% watched the recording back. These distributions also hold in the rest of the table.

The answers on the other questions will be discussed in the remainder of this section.

	Live present	Watching livestream	Watching recording	Total
AI	10	3	3	16
CS	13	4	4	21
Other	1	0	1	2
Total	24	7	8	39

Table 6.3: Distribution of students in the lecture

How did you experience the explanations of the design patterns?

Overall, the students were very positive about the explanations. They thought it was clear, easy to follow and interesting. Specifically, the mentioned that the used examples were very useful for their understanding and it helped that they were real life examples. However, 5 students indicated that they experienced the material as complex, vague and abstract, but this was mainly due the subject itself. Other remarks were that there was too much information on one slide and therefore, the recording was hard too read. Unfortunately, it was difficult to solve this issue, as there are already big UMLs on the slides due to the nature of the presented material.

There were no differences between AI or CS students and no differences between the ways the students saw the lecture, which means that the students experienced this explanation of design patters positively on all platforms.

How did you experience the used UMLs in the slides?

Again, the students were very positive about the UMLs used in the slides; they really helped with understanding the design patterns and most students were not overwhelmed by them. Students did indicate that they did not have a lot of experience with UMLs, but this was for most students no problem. 4 students stated that the UMLs were complicated at the start of the lecture, but once they were discussed in more detail, they could follow along. However, 2 students said that the UMLs were too complicated, mainly because they were overwhelmed by them and they were discussed too fast. Notably, some students indicated that they became a lot more familiar with reading and analysing UMLs as an additional learning effect.

We can see that the AI students struggled a bit more with UMLs than CS students; AI students thought the UMLs were more complicated and vague. Interestingly, the students that were watching the livestream live stated that the UMLs were overwhelming and that they were discussed too fast, while the students that were live present or watched the recordings did not mention this. But generally, there were no significant differences.

Could you focus this lecture?

27 students (69%) could focus easily most of the lecture and 12 students (31%) could focus around 50% of the lecture, which are positive numbers for a lecture.

There is a minor difference between AI and CS students; 63% of the AI students could focus easily most of the lecture compared to 76% of the CS students. When looking at the ways the students watched the lecture, there are interesting observations, as seen in table 6.2.

	Live present	Livestream	Recording	Total
Focus: Most of the lecture	15	6	6	27
Focus: Around 50%	9	1	2	12
Total	24	7	8	39

Table 6.4: Distribution of students' focus in the lecture

Surprisingly, almost all students watching the livestream and the recording could easily focus most of the lecture, while approximately a third of the present students could focus around 50% of the lecture. We expected that these distributions would be the other way around, but apparently students find it easier to focus when watching this lecture digitally. So, the structure of this lecture is an effective structure for students that are watching the lecture digitally.

How did you experience the exercises in the lecture?

Students were very positive about the exercises in the lecture for two reasons. Firstly, the exercises helped the students with understanding the material; not only could they test their understanding of the material, but they also started to think critically about it. Moreover, the students saw many examples of potential use cases which was beneficial, but also served as a helpful bridge towards practice. Secondly, the exercises motivated the students to pay attention in the lecture and created an active learning environment; they delivered the necessary interaction, engaged the students, but created some responsibility for the students as well. Furthermore, some students found the exercises too easy and some too difficult, but these balanced out pretty well.

There were no differences between the different studies and the ways the students watched the lecture. This means that whether students watched the lecture live (in the lecture room or via livestream) or the recording back, they all experienced the exercises as helpful and motivating, which makes these exercises a good feature to keep in digital teaching.

How did you experience the programs presented in Netbeans instead of code snippets on the slides?

Around 30% of the students mentioned that the programs in Netbeans were helpful and easy to follow, but not necessarily better than code snippets on slides. However, 40% of the students stated that the programs in Netbeans were better than the code snippets on the slides. The main reason given was that the students had a better idea of the actual structure and the implementation of the whole program, which is hard to grasp when providing code snippets on slides. Moreover, seeing actual, written code helped to translate the presented theory into code, which was a problem in the literature. The coding environment also helps to see the actual coding process, the debugger used to present the program interactions was also beneficial and the switch to another medium made the presentation more lively. On the other hand, around 10% of the students stated that code snippets were better than the programs in Netbeans, mainly because students experienced it as a bit chaotic and it was quite overwhelming, as students see all the details in the code while code snippets highlight the most important parts. Furthermore, it makes studying more difficult and it was not always good visible. In sum, the students were divided, but most of the students found it useful and we can use the given critics in the next iteration.

From the different studies we saw that AI students were more negative about the coding sessions, while the CS students thought it was better. There were no differences between the ways the students watched the lecture.

How did you experience this lecture compared to the other (Object Oriented Programming) lectures? And why?

25% of the students stated that the lecture was good, but not necessarily better than the other lectures. 50% of the students indicated that this lecture was better and clearer than their other lectures, but was also refreshing, which had several reasons. The main reason was that this lecture had more interaction and students felt more involved, which made the lecture easier to follow. This was mainly due to the exercises in the lecture, but another given reasons were the different mediums used in the lecture, e.g. exercises in an online quiz and programs presented in NetBeans. Moreover, the examples used were interesting, real-life examples, but not too difficult, which also helped with keeping attention. However, a single student mentioned that he/she preferred the "normal" lectures, but did not elaborate on this.

Looking at the different studies, AI students stated that this lecture was not better than other lectures, while half of the CS students indicated that this lecture was better and they felt more involved than other lectures. The students that were present live at the lecture were generally positive about the lecture and most students that stated that they were more involved fell in this group. Furthermore, 75% of the students that watched the recording back indicated that this lecture was better than their other lectures, which makes this lecture structure effective for later reference as recordings.

Other comments/remarks/?

Other comments were:

- The pointer was not always good visible on recordings
- The slides were relatively cluttered with text (unfortunately, the slides are already cluttered by the UMLs)
- It would be nice to show the given answers of the quiz
- Missing comparison of the patterns (was in the slides, but there was not enough time to discuss this)

Summary

Overall, students were very positive about the given lecture. Reasons for this were the clear structure of the slide set, the exercises in between the material and the live coding sessions. These elements led to an interactive lecture in which students could easily focus and therefore, everything was easier to follow. Students were also not overwhelmed by the UMLs, as these were discussed step-by-step, and the clear, but practical examples helped to understand these patterns better. Furthermore, the exercises not only tested the students on the actual structure of a discussed pattern, but also focused on the application of it, which captures both the learning objectives of this design pattern lecture. Moreover, all these different exercises gave even more practical example programs in which the pattern could be applied. Combining the UMLs for a clear overview of the patterns with a detailed implementation in the coding sessions and many different application examples in the exercises lead to a positive learning experience. Interestingly, there were no big differences between the students that were present live, were watching the livestream or watching the recording back, so the specified structure for this lecture could be effectively applied in such an environment. The given critics by the students are revisited in the discussion (chapter 8.5), in which we will discuss a potential next iteration.

6.3 Interviews at the practical session

Within the practical sessions, we talked to 20 groups of Computing Science students, 7 groups of Artificial Intelligence students (a total of 27 groups) and one Teaching Assistant. We categorised the results under comments about part 1 of the assignment, part 2 of the assignment, the designed lecture, the tutorial session and some general remarks.

Part 1 of the assignment

Around 35% of the groups did not encounter any problems in this part of the exercise and around 25% encountered minor problems. Even though 20% stated that the assignment was clear, there were a few groups that found the assignment unclear or too difficult. The main problem they encountered was that it was pretty clear which pattern they had to use, but actually implementing this pattern was the point where they got stuck. The slides and the Teaching Assistants helped the students to solve these problems.

Comparing the types of students, the AI students found this part a bit more challenging compared to CS students, but this difference was rather small.

Part 2 of the assignment

Many groups were still working on the second part of the assignment, so we collected less data on this part. In the results, 1 group encountered no problems, 2 groups found the exercise doable and 3 groups indicated that the assignment was quite clear. However, 2 groups stated that the assignment was unclear, 4 groups got overwhelmed by the assignment, 2 groups stated that the exercise was too difficult and 2 groups indicated that the second part was significantly harder than the first part. The main problems encountered by groups (so far) were: not enough experience for implementing the Binary Operator with an enumeration (5) and a Standard Library (5), using the Strategy pattern inside the Binary Operator (4), the generics that had to be used (3) and the Visitor Pattern itself (2). Other problems that were encountered were the details inside the Precedence function for the ShowVisitor and problems with the structure of the Formulas and the Boolean Connectives used in these Formulas. Overall, students understood the concept of the Visitor Pattern, but implementing the details was more challenging. To tackle these problems, students asked help from TAs and looked at the provided slides of both the tutorial session and lecture. Furthermore, students indicated that it would help if the exercise contained more information on the details of the implementation (the enumeration or the Standard Library), if the assignment was split in smaller steps and if there was a code template provided instead of having to create all the classes themselves.

Comparing the types of students, the AI students had a considerably harder time with this part, as many AI groups were still working on the assignment, while CS students were finished. Moreover, it were mostly AI students that indicated that the assignment was too difficult, that they got overwhelmed by it and that it would be nice to have a code template, so the start would be easier. The TA also agreed with the aforementioned points.

Lecture

From a few groups, we got some feedback about the lecture. Most of the comments were about the usefulness of the quiz used in the lecture for the interaction and the exercise. Other comments were related to the helpful structure of the lecture, the useful coding sessions and the useful UMLs used, which all made the lecture clearer. There were also several mixed opinions: some groups preferred the provided code within projects that was complete, while others prefer different code fragments in the slides. Furthermore, some groups indicated that the slides were clearly connected to the assignment, while some stated that the difficulty of the assignment was much harder than the content in the slides.

Tutorial session

Even though it is not within the scope of this research, we got some feedback on the tutorial session of this week. This feedback may be useful when constructing an extra lecture or tutorial session for a next iteration of this research. On one hand, students mentioned that the tutorial session was useful and the shown examples were good, but on the other hand, groups stated that the assignment was too complex compared to the content in the tutorial session. The feedback of the students was the following:

- Provide code or concrete hints for the more difficult details in the assignment (for this week, code for the enumeration and the Standard Library for the implementation of the Binary Operator for example) while still giving enough elaboration for the given code.
- Students would like more interaction in the tutorial session, which is pretty hard to achieve in our experience. One suggestion from the students was to let the students write code themselves in the tutorial session.
- It would also be helpful to focus on the problem solving part and understanding of the assignment instead of providing and discussing code to the students. As can be seen from earlier observations, many students struggled with actually understanding the assignment, so discussing the assignment in the tutorial session and breaking this up in smaller parts, would be very beneficial for the students.

In the results of the questionnaire after the exercise, we got a lot of positive feedback on the assignment. With the interviews, we could directly ask the students for more feedback, so we could get a more detailed opinion of them.

General remarks

The Teaching Assistant stated that the assignment was too big, but also too hard for the students. This problem could be tackled by removing the obligatory enumeration and Standard Library part from the Binary Operator, or at least providing proper support for this.

Many students could not finish the assignment in the scheduled practical session, which is often the case, so this also supports the previous statement. Interestingly, AI students found the assignment a lot harder (as indicated by other results in this research) and this resulted in an atmosphere in the room in which a lot of students were not focusing.

Lastly, one group noted that it would be more interesting to have an exercise about an authentic, real world example.

Summary

Overall, the assignment was too long and too difficult, according to the students and the Teaching Assistant. Furthermore, the Artificial Intelligence students experienced the assignment a lot more difficult than the Computing Science students, as in line with the other results of this research.

In more detail, part 1 of the assignment was relatively easy for the students; finding the correct pattern to use was doable, but actually implementing it caused some more problems. However, part 2 was significantly harder than the first part. The pattern itself did not cause the most problems for students, but this time the other details in the program that were mostly not directly related to the learning goals. These were for example the used enumeration, Standard Library and the Strategy pattern for the Binary Operator and the generics used in the ShowVisitor. A solution would be to completely remove these aspects from the exercise, or at least provide proper support and elaboration for them. Moreover, the assignment was also unclear for several students. Solutions for this would be more elaboration on the assignment or splitting up the assignment in smaller, clearer steps. This was also a suggestion for the Tutorial session, to make this session more effective.

Furthermore, students were positive about the lecture due to the added elements in our design (e.g. online quiz, the structure and live coding sessions).

6.4 Screen recordings of student groups

We received 3 screen recording of student groups while they were working on the assignment. Unfortunately, not all recordings were correctly recorded or complete, but there were still some interesting things to see in these recordings.

Part 1 of the assignment

The first thing we noticed was that the students were unsure how to start on the assignment, as it was the first refactoring exercise they made. Once they understood what was expected of them, creating the classes they needed and the interface was pretty doable. The only minor problems they faced were the typing they needed for the different functions and which code should be in which class. However, when the students had to implement the actual call to this interface and its classes, the students really struggled with the code they had to write, which was also the last segment of this part. After looking at the slides and some trial and error, the student figured out how to do it.

Part 2 of the assignment

At the start, students were clearly struggling with reading and understanding the actual assignment. Students had to create a UML of the program as the first exercise, which was the Visitor pattern. The pattern itself was quite clear, except that one group switched around the Visitors and Visitables. Moreover, the global UMLs looked decent, but many details were missing, e.g. how the binary operator is actually implemented and which attributes and function the classes/interfaces include.

When implementing the Visitor pattern, the groups did pretty well; implementing the Visitors was doable and the recursive calls in these were also correct. However, students struggled a lot with writing the correct code for creating and calling the implemented Visitors and Visitables, similarly to the Strategy pattern in part 1.

Moreover, the Strategy pattern inside the binary operator was challenging and difficult for the students to implement and use correctly; within their finished program, the pattern was often not correct and students found another, indirect solution to get the desired functionality.

Apart from these problems, other problems lay in the details of the program. One main problem was the propositional logic in the formulas, for which the students did not have the presumed knowledge, especially using the precedence values to determine whether the brackets should be placed or not. Students googled these definitions and to get all the related test cases correct, they simply tried to fix the cases with trial and error. Furthermore, students had problems with the used generics inside the assignment used in the Visitors and they did not know how the generic Map functions in Java, which was needed in the Evaluatorvisitor. Also, it was unclear what the expected result was of the ShowVisitor (e.g. AND operator is written as "/\" and not "&&"), which is easily solved by adding this to the assignment.

Lastly, students were asked to draw a sequence diagram, but all students we observed did not know what this was, so they had to google it and none of these groups submitted an adequate and finished sequence diagram.

General remarks

One interesting observation was that whenever the students had an error in their code, the students did not first look at the error, but were blindly adding code to fix the bug (e.g. adding random brackets and type castings). Furthermore, they immediately used the suggestions of the IDE to solve the bug, which were often incorrect and even more code was added, which made the problem only more difficult to solve.

To solve the problems the students encountered in the assignment, they used the provided slides, but also the code projects presented in the slides, to see concrete code as inspiration. Additionally, to see whether their program is correct and what the potential bugs were, the students effectively used the given test cases.

Summary

The first thing we saw was that students struggled with understanding the assignment and what they had to do. Afterwards, the assignments were doable and the main problems were not related to the discussed patterns, but to the other details of the program. How the patterns work, was pretty clear to the students (except the needed calls to actually use the patterns were difficult for the students), but the most time was spend on, for example, the functionality and usage of the precedence of propositional formulas, generics (using a Map as environment) or making a sequence diagram, while they had never done that. Specifically in part 2, students did not understand the underlying Strategy Pattern for the binary operator and no group had a solution that implemented or used this pattern correctly.

To solve the errors that occurred, most students did not think explicitly about the bug or error message, but made several blind and impulsive changes to the program, which did not fix the bug and sometimes only made the bug more complicated. This was only strengthened by the fact that students used the automatic bug solver in the IDE, which did often not suggest the correct code snippets. Additionally, the students used the presented slides, test cases and the provided code projects of the lecture to solve the problems they faced in the assignment.

6.5 Questionnaire after the assignment

The questionnaire was filled in by 117 students. With the first question, we could differentiate between the different studies: 59 students studied Computing Science (CS) (50%), 53 Artificial Intelligence (AI) (45%), 2 Physics, 1 Mathematics, 1 Pedagogical Sciences and 1 was a student from a high school. It was interesting to see whether there were any significant differences between the CS and AI students. The results of the questionnaire are discussed in the remainder of this section.

How did you experience the difficulty of the assignment?

The results can be found in table 6.5.

	CS	AI	Other	Total
Easy	0	2	0	2
Good	17	7	1	25
Challenging	32	34	3	69
Too difficult	10	10	1	21
Total	59	53	5	117

Table 6.5: Perceived difficulty of the assignment

The assignment was slightly too challenging looking at the results; 21% stated that the difficulty was good, 59% stated that it was challenging and 18% stated that it was too difficult. The reasons for the experienced difficulty are discussed in the next questions. Furthermore, AI students experienced the assignment as a bit more difficult than the CS students, but this difference is not significant.

Which problems did you encounter in the first part of the assignment?

38 students (32%) did not encounter any problems in the first exercise, where they had to refactor a given program with the Strategy Pattern. 17 students (15%) did only encounter minor problems (e.g. syntax problems), which were not worth mentioning and were solved pretty easily. Furthermore, the following problems were mentioned by the students (including how often it was stated):

- Requirements unclear of the assignment (22)
  It was unclear for the students what as expected from them from the exercise and therefore, it was hard to start the assignment. Reasons for this were that the "explanation was vague", but students also struggled with the idea of refactoring a program, as that was pretty new for them. As a result, students that understood the requirements were unsure which files they could remove or edit and whether they had to add more files.

- Recognising the design pattern (11)
  Several students struggled with recognising which design pattern was

the best to apply in the given program, but many succeeded in spotting the pattern eventually.

- Implementing the design pattern (10)
  After recognising the pattern, several students struggled with the details of the implementation. Examples of these are which are the Interfaces and which are the Classes, and placing the provided code in the correct Classes. Also, gluing all these different Classes together to make the pattern work was a challenge.

- Not enough knowledge of the subject of the program (6)
  Students were confused about the functionality of the `shiftNplaces` method (a method that takes a character and an integer n, such that the character ASCII value gets shifted n places). Moreover, students were confused with the idea of an encryptor and the given encryption methods, simply because they have not seen something like that at this point in their study. So, we had a misunderstanding of the base knowledge of the students in this field, so more instructions should be provided in the assignment to provide more explanation.

- Test cases not working (6)
  A few students had problems with the provided test cases. All these students used IntelliJ, but the test cases were written in NetBeans, so there was probably something wrong with the used version, but this was easily fixed.

- Factory unclear (2)
  A factory file was provided in the assignment for the test cases without any explanation, because we assumed that students were familiar with this. However, this was no the case for everyone, so once again, a bit of explanation should be provided for a given factory.

These problems were mainly solved with the help of the Teaching Assistants (TAs) (22), by looking at the slides (7) (mainly for deciding which pattern to use) and asking on the Discord server (2) or the internet (1).

A few other comments were given: a few students mentioned explicitly that this was a good exercise for understanding the usage and usefulness of a design pattern, but also engaged their critical thinking skills, which was missing in the other weeks. One students mentioned that he felt forced into the Strategy Pattern; we agree, but thought it was necessary, because the assignment would be even bigger if this task was more complicated.

From the results was also clear that the AI students struggled a lot more with this task compared to the CS students; 24 CS students encountered no problems compared to the 13 AI students. Moreover, 13 AI students did not understand the requirements compared to 4 CS students, so this is a significant difference.

Which problems did you encounter in the second part of the assignment while constructing the UML?

32 students (27%) did not encounter any problems in the second task where the students had to make an UML of the given program. 11 students (9%) only encountered minor problems, as the structure became more clear as the UML was drawn. 10 students (9%) reused the assignment of last year, so we did not get any data of these students.

Furthermore, the following problems were encountered:

- Applying and understanding the visitor pattern (20)
  This is where most students were struggling; the program in the task was clear, but applying the Visitor Pattern was more challenging. More specifically, students did not know what was supposed to be a Class or an Interface and which Objects should implement something (8). The individual components were clear to the students, but the whole structure of the Visitor Pattern not. Furthermore, it was not clear what the difference was between visitors and visitables (2) and how exactly the `visit` and `accept` functions were related to each other. Interesting to note here is that students struggled with the UML, because they did not yet have a complete overview of the program. So, it was not completely clear to the students that a complete overview is not yet needed for drawing the UML, as the process of drawing that helps to get the complete overview.

- Assignment hard to understand (9)
  The program was quite large and there were a lot of components to it, so students felt overwhelmed and the assignment was hard to understand. Moreover, the provided code to the students was minimal (especially compared to earlier assignments), so students were also missing a base to start from. We acknowledge these issues and for the next iteration, it is wise to provide some support for the students to understand the assignment, e.g. explaining the complete assignment in a lecture.

- Details in the UML (7)
  Some students understood the program and the application of the Visitor Pattern, but still struggled with the contents of the Classes and Interfaces, e.g. which functions should be where.

- UML problems (7)
  Students struggled with the UML syntax as well. For example, students did not know how to express generics in UML (3) (which is indeed not explicitly documented) or they did not have seen enough UMLs to draw one themselves, thus it took some time to get started (4).

- UML in slides not directly applicable in the assignment (7)
  The UML for the Visitor Pattern in the slides was a generalised version that contained an abstract Object which was implemented by all the visitables. However, this abstract Object can be ignored and was not needed in the assignment, but many students used the presented UML in the slides as a basis and were confused about this additional interface. For the next iteration it is an idea to change the presented UML in the slides or make a comment in the assignment about this.

- Technical problems in the programs for drawing UMLs (4)
  Several students encountered technical problems when drawing UMLs online, but these were easily solved.

These problems were mainly solved by looking at the slides (10) as these contained the UML of the Visitor Pattern, by asking TAs (6) and searching on the internet (4) or the provided program in the lecture (1). However, we observed that many students struggled with drawing a UML and decided to program the task first and make the UML afterwards (7). This is of course not the idea of a UML, so it could be emphasised more in the next iteration why making a UML is actually useful and helpful.

Comparing AI and CS students, there are a few interesting observations: firstly, more AI students (5) experienced issues with understanding the assignments then CS students (2). Moreover, CS students had more struggles with the UML than AI students which was seen in different areas; confusion about the abstract Class above visitables (4 CS, 2 AI), problems with the details in the UML (6 CS, 1 AI), program structure was unclear (8 CS, 3 AI). What we expect, but this could to be validated by the students' handed-in work, is that the UMLs of the CS students are more detailed than the AI students, as the CS students experienced more, but also more detailed problems. We can check this by comparing the work of the AI students to the CS students, but unfortunately, we did not had the time for this.

Which problems did you encounter in the second part of the assignment while implementing the formulas (the visitables)?

24 students (21%) did not encounter any problems in the second task where the students had to implement the Formula structure, which are the visitables. 17 students (15%) only encountered minor problems, such as forgetting keywords or using an incorrect string representation for the boolean connectives. Again, 10 students reused the assignment of last year.

Furthermore, the following problems were encountered:

- Problems with implementing the Binary Operator (20)
  In the assignment, the boolean connectives were represented by a single class with `BinaryOperator` as field, thus being constructed as a

Strategy Pattern within the Visitor Pattern. However, not much information was provided on how to apply this Strategy Pattern and thus, students were experiencing problems with this application. Moreover, the students were asked to use a Standard Library for this implementation. There was also not much information provided for this and students did not do this before, so a few struggled with this as well. For the next iteration, it is wise to provide some more guidelines for these things or completely remove them from the assignment.

- Problems with implementing the enumeration for boolean connectives (19)
  The boolean connectives ($\wedge$,$\vee$ and $\rightarrow$) had to be implemented using an enumeration type. However, students did not have much experience with such an enumeration and very few guidelines were provided, so students did not know where to start here. Furthermore, combining this already challenging enumeration type with the Strategy Pattern used to represent Binary Operators was very difficult for students and took a lot of time and energy from them. This part of the assignment should be explained in more detail or should completely be removed, as at the moment, the students are putting too much time in this, while it is not the most important part of the program.

- Assignment was unclear (12)
  There was a lot of information in the assignment and students were overwhelmed by it. It was unclear what was expected from the students and what they had to exactly implement. The difficult part is that all the information is necessary, but it is possible to reorganise this information in a more structured manner.

- Starting with the assignment was hard (12)
  Once the students understood what should happen, it was still hard to actually start with the assignment; there had to be implemented so much that there was no clear point where to start. Afterwards, the logical order of implementing parts of the program was also unclear, so it could be an idea to provide some more guidelines here for the students, such that they are not stuck at the start of the implementation.

- Problems related to the Visitor pattern (9)
  There were several problems related to the application of the Visitor Pattern: the difference between visitors and visitables was not clear to the students, the purpose of different functions was not clear (e.g. `accept`, `visit` and `getPrecedence`) and where certain Interfaces and Classes were needed. We expected these problems, as it was the first time students implemented this pattern and these are the most challenging aspects.

- No clue what they were doing (7)
  Some students stated that they had no idea what they were doing and were not even able to specify which problems they faced in the assignment.

- Specific problems

  - Some students were not familiar with propositions and proposition logic (2)
  - It was not clear what the precedence was of boolean connectives and how this was supposed to work (1)
  - One student struggled with the Lambda expression (which was not necessary to use)

These problems were solved by asking TAs (16), looking at the slides (13), looking at the internet (1), Discord (1) or the provided test cases (1).

Comparing AI and CS students, CS students struggled more with the Strategy Pattern of the Binary Operator (13 CS, 5 AI) and the enumeration (15 CS, 3 AI). However, this is probably due the fact that CS students were able to specify these problems as many AI students couldn't specify their problems (2 CS, 5 AI) and 9 AI students reused their assignments. Moreover, significantly more CS students used slides to solve their problems compared to AI students (10 CS, 2 AI).

Which problems did you encounter in the second part of the assignment while implementing the visitors?

17 students (15%) encountered no problems, 30 students (26%) only minor problems, such as variables that were switched around and minor bugs. Once more, 10 students reused the assignment of last year.

Furthermore, the following problems were encountered:

- Problems related with the ShowVisitor (25)
  Many students struggled with the implementation of the ShowVisitor, which had to print the given formula with the correct and needed parenthesis. All these problems were related to printing the parenthesis, which could derived from the precedence value of the boolean operators. We assumed that all the students were familiar with the idea of the precedence values and how this helps to decide which parenthesis are indeed needed in the formula. So, the precedence was mentioned in the assignment, but the details were left unspecified, with the expectation that the students had enough knowledge about this, but this was not the case. Many students struggled with the usage of the precedence and how this helps to print the parenthesis and as a result, many students took a long time to implement this or were not

able to do it at all. As we had the expectation that students did not encounter any difficulties here, we thought it would be a good addition to the assignment, but in hindsight, we should provide more information or remove the parenthesis completely from the assignment. The focus of this assignment lies on the Visitor Pattern and not the details of implementing a correct system of printing parenthesis.

- No clue what they were doing (10)
  There were also students that had no idea what they were doing in this part and they were not able to specify explicit problems they encountered.

- Problems related to the Visitor pattern (8)
  These were mainly problems related to the connection between the `accept` and `visit` functions and how these should exactly be implemented. Moreover, it was confusing at first how to execute a recursive call with the `accept` function, as this is not immediately intuitive.

- Starting was hard (6)
  Some students found it hard to start this section, mainly because it was not immediately clear how all the Classes and functions were supposed to work together. Luckily, after the students looked at the Visitor Pattern again, it was quite clear how it was supposed to work.

- Problems in the Factory (2)
  Students did not know how to exactly call a Visitor from the Factory and other students forgot to implement the Factory, so their test cases wouldn't work, but these problems were easily fixed.

- Problems related with the Evaluator (2)
  A few students struggled with the Evaluator instead of the ShowVisitor. This was mainly due the formulation of the assignment, so it was unclear what the Evaluator was exactly supposed to do.

- Problems with the generic typing (1)
  These students struggled with the generic typing that was used in the assignment.

These problems were solved by asking TAs (10), looking at the test cases (4), asking the Discord server (2), or looking at the slides (2) or the internet (1). There were no significant different between the CS and AI students in these results.

Did you encounter any other problems?

Students mentioned a few other problems they encountered:

- Students never saw a sequence diagram before (8)
  The students had to make a sequence diagram of the program, but they had never seen such a thing before. Therefore, the students were very confused what was asked of them.

- The assignment was too hard to understand on you own (5)
  Students mentioned that the assignment was too hard and too long and as a result, it was for them impossible to make it without help from the TAs.

- The tree structure used for the Formulas (1)
  These students struggled with the tree structure used to represent formulas.

Comparing the CS and AI students with each other, the AI students had more trouble with understanding the assignment (1 CS, 4 AI), which was to be expected, as the AI students struggled more with the assignment looking at the other results. Furthermore, the CS students complained about the sequence diagram (7 CS, 1 AI), so we expect that the AI students saw a sequence diagram earlier in their courses.

Were the slides useful for this assignment? Was there a clear connection between the taught material and the assignment?

Regarding the connection between the slides and the assignment, students were very positive. 8 students indicated that the slides were helpful and 63 students stated that the slides were useful for making the assignment. 22 students indicated that the slides and assignment were clearly connected and thus, the material in the slides could almost directly be applied in the context of the assignment. However, 12 students stated that the assignment was significantly harder than the material covered in the slides and the explained material did not help the students in this assignment. Also, 2 students mentioned that the details of the assignment were not covered in the slides which took a long time to figure out themselves. For example, a reference for the usage of enumerations or Standard Libraries would have been very helpful.

Moreover, students also gave specific feedback on the slides themselves, which are in line with our earlier observations. Reasons for this were the clear, step-by-step structure used in the slides (3), the useful examples (2), the exercises in between the covered material (6) and the coding session with the provided programs (5). Also, 13 students stated that the UMLs were very helpful; not only for their understanding, but also for making the assignment. Additionally, 2 students specifically mentioned that the given references at the end were very useful for understanding the discussed patterns and were helpful for the assignment. However, there was also some critique on the presented slides; e.g. 5 students mentioned that the useful information in the slides was cluttered by exercises and that a better balance should be found for information on slides and exercises. This is easily solved by providing an additional slide set to the slides in which the exercises are removed. Students also found the slides a bit confusing sometimes due to the discussed material (3) and found it hard to actually translate the given information into code (3). Lastly, students stated there were too many examples given compared to the provided code (1) and students still struggled with the UML syntax in and after the lecture (1).

Students also provided feedback on the tutorial slides provided in the assignment. 14 students found the tutorial slides very helpful for this assignment and 2 students specifically mentioned the provided code snippets on the slides. Still, 1 student stated that the tutorial slides were a bit chaotic and thus not usable for the assignment.

Comparing CS students and AI students, there were no notable differences here.

Other comments/remarks?

There were a few positive remarks here: students mentioned that the assignment significantly helped with their understanding of the material (1), the assignment was fun to work on (1) and the topic was interesting and applicable in a broader sense (1).

There was also some criticism on the assignment:

- The assignment description was too vague (5) and too chaotic (1). This can be solved by structuring the assignment a bit more.
- The assignment was too big and took too much time to make (5).
- The first part was too hard (2). This is interesting, as many students faced problems in the second part of the assignment.
- The slides had not enough information in them to make the complete assignment (6). This mainly entails the details in the code that were not provided anywhere (the enumeration or the binary operator), so students had to google it while the provided slides were of little use.
- The assignment mainly focused on code details instead of the actual pattern (1). For example, a lot time was spend on fixing the parenthesis in the `ShowVisitor`.
- There were typos in the assignment (2).

Lastly, there were also a few suggestions which could fix these issues: it would be fun to have a real world example to work on (1), it would help if there was more code provided to the students (1) and it would be useful if the whole assignment was discussed in the lecture/tutorial or a similar example of the assignment (1).

Comparing the AI and CS students, the AI students struggled significantly more with this assignment. This can be seen from the comments that the assignment was too big (1 CS, 4 AI), too vague (0 CS, 6 AI) and that the first part was too hard (0 CS, 2 AI)

Summary

The first part of the assignment was about recognising a pattern and refactoring the program. Around 50% of the students encountered no or minor problems. The other students experienced that the requirements were unclear of the assignment and other problems were recognising and implementing the design pattern. These latter two were expected, as these were part of the learning goals and the former can be easily fixed by updating the requirements and making them more explicit.

The second part was about applying the Visitor pattern. Around 30-40% of the students encountered no problems or minor problems in this part. Other students were still confused by the Visitor Pattern, mainly the relation between visitors and visitables and how the `accept` and `visit` functions connected these two. These problems still occurred later in the assignment, but were mostly fixed after constructing the UML in the first subquestion, as the students had to actively think about the pattern here. Furthermore, the students did not necessarily struggle with applying the pattern, but most of the time with the details of the program, as will be discussed later.

Students also had some comments on the assignment in general. One thing we saw was that, even though this was the first time students actively saw UMLs and had to construct them, students understood how they worked and functioned, so that was another benefit of the lecture. However, the difficulty of the assignment was a bit too difficult, which can be fixed by giving a more structured explanation of the assignment and more instructions on the challenging details. For example, many students struggled with the Binary Operator, which had to be implemented via an enumeration and a Standard Library. Students did not have to do this before, so many spend a lot of time on this, while it was not the key part of the assignment. Similarly, students did not understand the precedence of the formulas, which is also easily solved by giving more hints here, instead of no explanation, which is the case right now. Moreover, students were asked to construct a sequence diagram, while they had never seen this and never made one. The main conclusion we can see here is that it is wise to think critically about the learning goals and try to remove the parts which are difficult for students and take a lot of time which are not really relevant for the learning goals. At least, provide enough instructions, such that students do not have to figure out everything themselves.

Comparing Computer Science students and Artificial Intelligence students, we saw that the Artificial Intelligence students experienced the assignment as a bit more difficult than Computer Science students.

We also saw that the designed lecture positively contributed to the assignment. When stuck, students mostly assessed the lecture slides, apart from asking help of the teaching assistant (Asking a TA 53x, Looking at slides 30x, Looking at internet 9x, Looking at the test cases 5x, Asking on Discord 3x). Additionally, around 75% stated that the slides were useful/helpful and that the slides and assignment were clearly connected with each other. Especially the UMLs were helpful in the slides and the other elements we put in our slides (exercises, the structure and the provided programs in the coding session). However, there was also feedback that the assignment was significantly harder than the slides and that important details of the assignment were missing (the aforementioned binary operator and enumeration). We can keep this in mind as feedback for the slides, but is even more useful in a potential tutorial session.

6.6 Exam results

We received 346 exam entries of the students, but 97 students did not fill in the exam question, so we got a total of 249 useful entries.

Visitables (exercises 1 & 2)

The distribution of the entries for exercises 1 and 2 are found in table 6.6. As the distribution is almost the same, we only made a pie chart of exercise 2, which is found in figure 6.1.

	Exercise 1	Exercise 2
Correct	69	66
Visitor call incorrect	90	90
Accept function completely incorrect	43	43
Accept function missing	30	30
Completely incorrect	16	20

Table 6.6: Distribution of exercises 1 and 2 of the exam

As we can see, 27% of the students had the exercise correct, and thus understood how to use visitables. 20% of the students probably had no idea what the Visitor pattern was, as their entry was incorrect or did not include an `accept` function. 17% of the students knew the outline of the Visitor pattern; they knew there had to be an `accept` function, but they

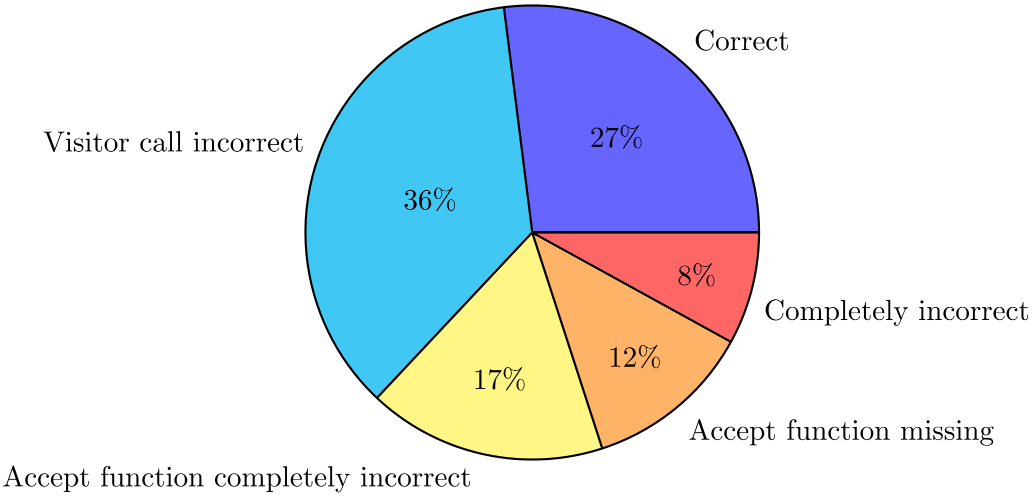


Figure 6.1: Distribution of exercise 1 of the exam

did not know the details. The other 36% were pretty close to a correct implementation, but made a minor mistake within the Visitor call, which could be categorised as follows (based on exercise 2, as the results exercise 1 were equivalent):

- Return was missing (55)
  `visitor.visit(this)` instead of `return visitor.visit(this)`
- Return this (8)
  `return this` instead of `return visitor.visit(this)`
- Copy of current is Object is passed as argument (5)
  E.g. `return visitor.visit(new Sum(p1, p2))` instead of `return visitor.visit(this)`, where `p1` and `p2` are the left and right "sub-polynomial"
- 2 calls for both the attributes in sum (4)
  E.g. `return visitor.visit(p1) + visitor.visit(p2)` instead of `return visitor.visit(this)`
- Visitor Object is returned (3)
  `return visitor` instead of `return visitor.visit(this)`
- And 15 other variations which did not fall into categories with 3 or more occurrences, for example infinite recursion or a missing argument in the `visit` call.

Most frequently, the students did not put a return in front of their visitor call. However, it could be argued that this should be seen as a correct answer, as it is a very minor detail, so then there would be 121 students with a correct answer (49%). From the other categories, we can deduce that many students knew there had to be a call with a `visit` function, but the exact coding was not known, even though this is standard boilerplate code.

So, we see that many students understood the structure of the Visitor Pattern (regarding the visitables), so they know there should be an `accept` function. However, students did not know the boilerplate code of the visitor call in this function.

Visitors (exercises 3 & 4)

The distribution of the entries for exercises 3 and 4 are found in table 6.7. As the distribution is almost the same, only a pie chart for exercise 3 was made, which is found in figure 6.2.

	Exercise 3	Exercise 4
Correct	50	51
Visit functions' body is incorrect	87	83
Visit functions are completely incorrect	40	35
Visit functions are missing	49	47
Completely incorrect	23	33

Table 6.7: Distribution of exercises 3 and 4 of the exam

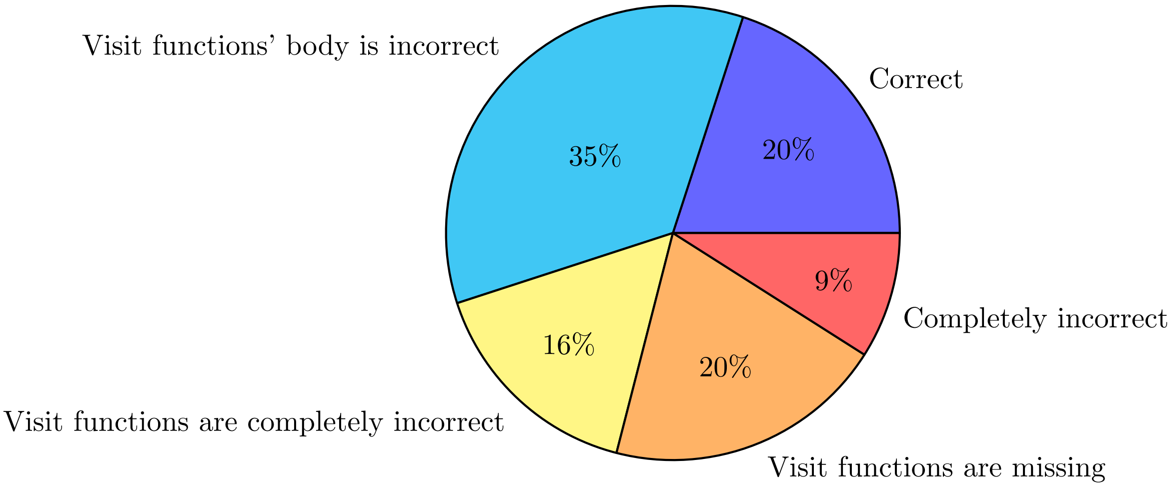


Figure 6.2: Distribution of exercise 3 of the exam

Compared to the previous exercises, less students made exercises 3 and 4 completely correct (20 %). 29% did not have the necessary knowledge of the

Visitor Pattern for the visitors, as 20% did not use `visit` functions and 9% had a completely incorrect submission. 16% of the students created `visit` functions, but the bodies were missing or the written code was not enough in line with the solution. The other 35% that had written `visit` functions that had minor mistakes, could be categorised as follows (based on exercise 3, exercise 4 was very similar):

- Recursive calls called `visit` instead of `accept` (68)
  Within the `visit` function for the `Sum`, students wrote
  `return visit(s.getP1()) + visit(s.getP2())`
  instead of
  `return s.getP1().accept(this) + s.getP2().accept(this)`
  where `s` is the provided sum polynomial and `getP1()`, `getP2()` are the getters for the left and right polynomial of `s`.

- Recursive calls are completely incorrect (9)
  This implementation of the `visit` functions had the recursive calls (or a variation of)
  `return s.getP1() + s.getP2()`
  instead of
  `return s.getP1().accept(this) + s.getP2().accept(this)`
  which simply would not work.

- `visit` function is only `accept` call (7)
  `return s.accept(this)`
  instead of
  `return s.getP1().accept(this) + s.getP2().accept(this)`
  where `s` is the provided sum polynomial.

As can be seen, a major fraction of these students thought they could use the `visit` functions instead of the `accept` functions, so it is probably not clear how the `visit` and `accept` functions are actually related.

So, we again see that many students understood the structure of the visitors in the Visitor Pattern, as they knew there should be `visit` functions and what their general functionality should be. Still, the exact calls that should be executed are not clear to several students, mainly the difference between the `visit` and `accept` functions and how they are related.

Using the Visitor Pattern (exercise 5)

The distribution of the entries for exercise 5 are found in table 6.8 and figure 6.3.

As can be seen, 32% of the entries had a correct solution and 29% had a completely incorrect solution. The other 39% had some minor mistakes in their function calls which were categorised as follows:

	Exercise 5
Correct	79
Function calls are incorrect	97
Completely incorrect	73

Table 6.8: Distribution of exercise 5 of the exam

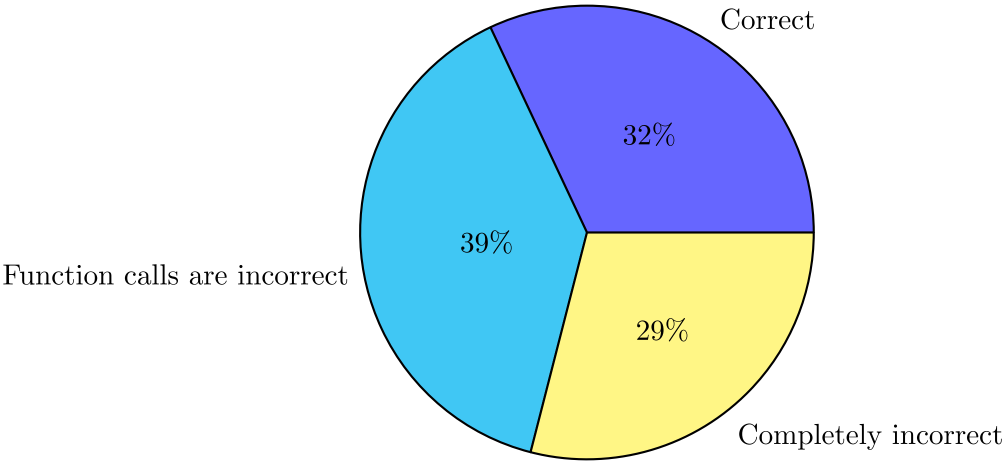


Figure 6.3: Distribution of exercise 5 of the exam

- `Visit` function is used instead of `accept` (58)
  For example, `showVisitor.visit(poly)` instead of `poly.accept(showVisitor)` where `showVisitor` is a showVisitor object and `poly` is the current polynomial.

- Polynomial is provided as an argument in the instantiation of the Visitor (23)
  `ShowVisitor sv = new ShowVisitor(poly);`
  instead of
  `ShowVisitor sv = new ShowVisitor();`.

- Visitor name as function call (12)
  For example, `poly.showVisitor()`, even though `showVisitor()` is not a valid function.

- Using non-existent functions within the Visitor (3)
  E.g. `showVisitor.show()` while `show()` is not implemented.

- `Accept` function used on Visitor Object (1)
  `showVisitor.accept(poly)` instead of `poly.accept(showVisitor)`

As can be seen from this exercise as well, students are not certain how exactly the `visit` and `accept` functions work and how they are related. Furthermore, we see that some students were not familiar with the exact attributes of the Visitor and the details on how to use Visitors on Objects.

Summary

From the entries of the exam results, on average 25% of the entries were fully correct, but 26% were completely incorrect. Of the other entries, we saw that the structure of the Visitor pattern was pretty clear, for example, students knew there had to be an `accept` function in the Visitors and a `visit` function in the Visitors. Only the actual attributes of the Visitors were not very clear to the students, as could be seen from exercise 5. Even though the structure of the pattern was clear, the details of the code within this pattern were more challenging. Examples of these are the boilerplate code for the `accept` functions, using non-existing functions within function calls and using `visit` functions in recursive calls instead of `accept` functions. The underlying problem for these observations is that the students were not familiar enough with the `accept` and `visit` functions; especially how the are relate, how they differ and when to use which one.

6.7 Triangulation

As there was some overlap in the previous results of the different data collections, we could triangulate this data. The observations we saw in different results are presented here, categorised under feedback on the lecture, feedback on the assignment and the students' understanding of design patterns.

Feedback on the lecture

- Students were positive about the lecture due to the added elements
  Students were positive about the lecture and found the slides useful. This was due the addition of the exercises in the form of an online quiz, the clear structure of presenting and discussing a pattern and the code projects used in the live coding sessions. All these elements were mentioned in the questionnaire after the lecture, the questionnaire after the assignment and were also seen in the interviews at the practical sessions.

- Students had a good understanding of UMLs after the lecture
  From the questionnaire after the assignment, we saw that the students did not encounter any problems regarding UMLs and that these were pretty clear to them. This could be a learning effect of our lecture, as the students mentioned they were not overwhelmed by the UMLs in the slides and we saw a learning process regarding UMLs in the results of the online quiz.

- Slides and presented code projects were used as a reference
  The students already mentioned that the slides were useful, but the students stated that they used the slides and code projects to start on the assignment and tackle problems. This was also seen in the screen recordings of the students, where they accessed the slides and imported and inspected the code projects while working on the assignment.

Feedback on the assignment

- Assignment was too hard
  Within the questionnaire of the lecture, we already got the feedback from students that the assignment was too hard. We also got this feedback directly from the students while doing the interviews at the practical session as well as that the assignment was unclear and it was hard to start. These things were also seen in the screen recordings we got from students, because students took a long time scrolling through the assignment and were struggling when starting with the assignment.

- Main problems were in the details of the assignment instead of the application of the pattern
  In the questionnaire after the assignment, we saw that the main problems were not related to the pattern, but in the details of the program. In this assignment, that were specifically the implementation of the Binary Operator which had to be done with a Strategy Pattern, an enumeration and the usage of a Standard Library. Other problems were the precedence of operators, generics and creating a sequence diagram for the first time. These problems were also seen immediately in the interviews at the practical sessions, but also in the screen recordings of the students.

Students' understanding of design patterns

- Structure of design patterns are clear, but implementation details are unclear
  We saw from the results that the structure of the patterns was pretty clear to the students, as they knew which Classes should be used and which are the main functions to make the pattern work. However, the implementation details are more challenging for the students. This was best seen within the Visitor Pattern, where the students were not certain about the attributes of the Visitors and its constructor. Furthermore, the students were uncertain of the difference between the `visit` and `accept` functions and how they are related. Not only did we see these results in the questionnaire after the lecture, but also in the observations at the practical session and the screen recordings. More-

over, these results were best seen in the exam entries of the students, where we took a closer look at the students' code.

Chapter 7

Conclusions

7.1 Research Subquestions

Research subquestion 1: How can the students' application and knowledge of design patterns be characterised after the learning activity?

The students had a high-level understanding of the UMLs of the design patterns; they knew which Classes and Objects were used and the necessary functions and attributes. Furthermore, the students were able to recognise in which programs design patterns could be applied and when patterns could not be applied, at least for the design patterns discussed in the lecture. It was also apparent that the students improved in these skills in the course of the lecture. Additionally, we saw that the students were getting more familiar with reading and creating UMLs during the learning activity. The students were also able to refactor assignments with a design pattern they had to recognise, but students struggled with the implementation details of the design pattern. These problems were related to the usage of the pattern once implemented and the details and interactions of functions used in a pattern. For example, students were able to implement the Strategy Pattern relatively easy, but the initialisation of the classes and the needed function calls to use the implemented Strategy Pattern were challenging for the students. Similarly for the Visitor Pattern, students had difficulties with the boilerplate code in some functions and did not completely understand the relation between and the usage of the `accept` and `visit` functions. Additionally, we saw that the exact function calls to use the implemented pattern were not clear to the students. So, we saw that students had a high-level understanding of the design patterns and their structure, but the difficulties lay in the implementation details of the patterns, especially the function bodies and calls within the patterns and the function calls to use the implemented pattern.

Research subquestion 2: How did the students perceive the designed lecture?

In general, the students were positive about the given lecture and the slides were often used as a reference in the assignment. The slides were also clearly connected to the assignment students had to make. Furthermore, the students indicated that the exercises within the lecture were beneficial for their focus and it stimulated active learning. Additionally, the many different use cases the students saw within these exercises helped the students to better understand the use cases of the patterns. Next to the exercises, students were more positive about the live coding sessions compared to code snippets copied on the slides. Students also mentioned that the lecture was interactive (mainly due to the exercises and live coding sessions), resulting in a better focus and material that were easier to follow.

The students found the clearly structured explanations of the design patterns helpful as well: the UMLs gave an overview of the pattern, the live coding sessions the implementation details and the exercises the practical use cases. This structure also ensured the explanation of the implementation details of a pattern as well as the practical use cases, and the students pointed that out as well. Note that the students were not overwhelmed by the shown UMLs for the design patterns. Moreover, there was no difference between the students that were present live at the lecture, watched the live stream or the recording afterwards.

7.2 Research question

How can a learning activity contribute to the students' understanding of design patterns?

Within the lecture of the learning activity, we explained a design pattern in a structured manner: start with a concrete problem that can be solved with the design pattern, discuss the abstract UML of the pattern, discuss a concrete UML of an application and the actual implementation in a live coding session and end with exercises with cases where the pattern should be applied or not. This structure leads to an interactive lecture and a lecture in which students can focus better, especially with the embedded active learning. Additionally, the recurring problem of the abstractness of design pattern is also successfully solved with the number of provided concrete examples. The two learning goals related to design patterns are both tackled: recognising design patterns in a provided program and implementing a design pattern. The UML provided a global overview, the live coding session provided a concrete implementation and the exercises afterwards served as several concrete examples of the application of the design pattern. These learning goals also come back in an assignment consisting of 2 parts: one refactoring exercise and one exercise in which a specific design pattern should be implemented. An additional learning effect was that the students became more confident with reading and using UMLs after this learning activity.

In the course of the lecture, students were getting more familiar with recognising patterns in given programs and also observing when a pattern should not be applied, even in more difficult examples. The accomplishment of the learning goal related to recognising design patterns was also recognised in the assignment, as students found the refactoring exercise relatively easy. The only problem they faced where the function calls needed to use the implemented pattern.

For the learning goal related to implementing a pattern, students encountered some difficulties. We observed that the students had a high-level understanding of the design patterns, i.e. they knew the underlying UML, its classes and functions, but the students had a hard time with the concrete function bodies and the relation between different functions. One example of this are the relation between the `accept` and `visit` function in the Visitor pattern. Additionally, similarly to the observation in the refactoring exercise, students struggled with writing the exact function call which was needed to use their implementation of the design pattern.

So, the learning activity contributed positively to the students' ability to recognise design patterns in program and applying them. Furthermore, the students had a high-level understanding of design patterns, but there were problems related to the function bodies of the patterns and the function calls needed to use these patterns. Adjusting the learning activity to focus more on these function calls could solve these problems.

Chapter 8

Discussion

In this chapter, we will discuss and reflect on various parts of this research. We start with discussing the conclusions, then we discuss other observations in the found results, followed by a reflection on the methodology. Afterwards, we compare our results with the findings in the literature and which problems were tackled. Lastly, we discuss a possible next iteration with several recommendations as potential future work.

8.1 Reflection on the conclusions

One important note on our conclusions is that we could state significantly more about the students' perceptions of the designed lecture compared to the actual learning effects. The reason for this is that in our research, it was difficult to verify what were the exact learning affects of our learning activity. The main issue here is the fact whether it is unclear if the knowledge of the students is indeed related to the designed lecture or the students in general, independent of the lecture. This could be verified with the usage of an experimental and control group, similar to Khakim (2019/08), or comparing the results with earlier years, similar to Xinogalos (2015). However, we decided that these methods were not viable for our research, but this is discussed in more detail in section 8.3.

Even though we could not validate the learning effects in the detail we wanted, we still expect that the students' application and knowledge of design patterns, discussed in the last research subquestion, are still mostly related to the given lecture, even though there is no hard evidence for this. The fact that the students had a good understanding of the applications of design patterns and a high-level understanding of the implementation of design patterns, but still struggled with the exact function bodies and function calls for the design patterns, can be with reasonable certainly seen as a direct consequence of the lecture. The reason is that the focus of the lecture was on the application part of the design patterns and the UML of the patterns, but the exact function bodies were not discussed in much detail. The function calls needed to use the implemented pattern were mentioned in the lecture, but overall little time was spent on this. These two points were discussed in the live coding session, but should in hindsight be discussed in more detail, to tackle these current observations.

Furthermore, as commonly seen in the literature, researches similar to this often have second iteration were they incorporate the findings of their first iteration and have another iteration which they can use for comparison. Unfortunately, such a second iteration was not possible in this research, due to the time constraints the the planning of the courses at the Radboud University. However, recommendations for a second iteration are discussed in section 8.5.

8.2 Additional results

Most of the results found in this research were used to answer the research (sub)questions, but there were also other observations which were not directly applicable for the research (sub)questions. The observations were related to the difference between Computing Science (CS) and Artificial Intelligence (AI) students, the differences between being present live at the lecture, watching the livestream and watching the recordings and lastly, other observations we saw related to the coding skills of the students. These results are discussed in the following sections.

Differences between Computing Science and Artificial Intelligence students

Related to the lecture, there were a few differences between the AI and CS students. Firstly, the AI students had a harder time focusing in the lecture compared to CS students. Additionally, AI students indicated afterwards that they did not found this lecture necessarily better than their average lectures, while the CS students were quite clear that this lecture was better. So, we expect that the AI students have "better" lectures in general, or at least different types of lectures than CS students, which would explain these differences. Furthermore, the AI students had a harder time understanding the UMLs of the design patterns and AI students were more negative about the live coding sessions, compared to the CS students who really liked these live coding sessions. An explanation here could be that CS have seen more code and have programmed more in general, so they could adapt more easily to the live coding session in a coding environment, while the AI students got easily overwhelmed.

There were also observations related to the assignment the students had to make, of which the main observations was that AI students had a considerably harder time with making the assignment than the CS students, which we could conclude from different results. From the questionnaire after the assignment, AI students indicated for both parts that they found it too difficult, especially compared to the CS students. The first part was a refactoring exercise, which could be easier for the CS students, as they have written, and thus refactored, more code and an explanation for the second part could be that CS students were more familiar with the subject of that exercise, namely binary formulas. AI students also indicated that they were overwhelmed by the assignment and that it would help if there was a code template with which they could start. Furthermore, the CS students could specify their problems a lot better than AI students, which was probably due the fact that CS had an easier time with this assignment, while the AI students were stuck at the start of the assignment. Another explanation could be the observation that CS students used the slides of the lecture significantly more than AI students, so perhaps they had a better guidance in the assignment due to the usage of these slides. Moreover, we saw that there were a lot of AI students that reused their submission from last year, which could again be explained by the fact that AI students do have a harder time with this course and this assignment. We also saw this immediately while doing the observations in the practical sessions, as there was less focus in the tutorial sessions of the AI students, as they were mostly confused with the assignment, which lead to less focus on the actual assignment.

Differences between the media used to watch the lecture

The main observations made here is that the designed lecture was applicable and effective for all the different media: being present live, watching the livestream or watching the recording afterwards. All the different media could easily focus in the lecture, even though we expected that the livestream or the recordings would struggle with this. Interestingly, all the different media were also doing the exercises in the online quiz in the lecture, so this quiz was not only taken by the students that were live present. The students that watched the recording also indicated that the recordings were easy to follow and helpful as a reference without being present live, so that means that these recordings could also be used in the following years. A few differences were that the students that were present live were more involved in the lecture (which is not surprising) and that the only group that were overwhelmed by the shown UMLs were the students that watched the livestream. The reason for this could be that the students that were present live were already more involved and that the recordings could be played back if the students got lost and the students that watched the livestream did not had any of these benefits.

Other coding related issues

When looking at the results, we saw that the students did not have the assumed knowledge about binary formulas, even though this topic is often covered in different courses. Additionally, even though this was covered a few weeks before the design patterns in the same course, students were still not familiar enough with the usage of generics, which was (apparently incorrectly) assumed that the students got these skills. On a more general note, students are having a hard time understanding a complete assignment and what they are exactly supposed to do and how to start, which they mentioned was also occurring in the earlier weeks. Furthermore, and perhaps even the most interesting, we saw the the error solving skills of the students were not of the level we expected. Apparently (as far as we could see), when students see an error in their code, they do not analyse it and think about it, but use the automatically generated suggested solutions to solve their errors, without thinking about the consequences and whether that is a correct solution. As a result, the students add lines to their program which were suggested automatically which are not always correct, and thus they make the debugging even harder for themselves. Therefore, a lecture focused on debugging could be wise, next to the live coding sessions which could also contribute to this observations.

8.3 Reflection on the methods

We mostly based our methods on the methods used in the literature and we experienced these data collections from different sources as extremely helpful and the triangulation of it. An example of this were the questionnaire after the assignment, the screen recordings of students and the observations: the screen recordings provided a source of a few students, but in a lot of detail, the questionnaire was a relatively high-level data source, but from almost all students and the observations were in the middle of these two. In hindsight, every data source contributed to the final conclusions of this research, so we would not have omitted one.

One note on our methods is that the results in the questionnaires after the lecture and the assignment could be biased, as the students could perhaps give the answer that the teacher or the researcher ”wants” or ”desires”. However, we do not think this is the case, as the results of the questionnaire after the lecture is in line with our expectations, which are the findings in the literature, and the results of the questionnaire after the assignment is in line with the observations in the screen recordings and the observations in the practical sessions.

There are some other methods which we considered to use in our research, but which were not feasible or probably not useful in hindsight. For example, we could have looked at the students' solutions to the assignment instead of the exam results. One reason why we chose not to, was that this assignment was made in pairs and students got a lot of help, which were both not the case for the exam submissions, so in that perspective, the exam submissions were more useful. Additionally, the assignment solutions would probably be working solutions, so firstly, it is possible that code is copied from others and handed in such that they have a working solution, and secondly, it would not be helpful to look at the handed-in solutions, as the problems that the students faced and the misconceptions would not be seen here. Therefore, the exam was, in our opinion, the better alternative of these. Note that we could also looked at the resit exam submissions, but this sample size would have been significantly smaller and we expected that these results would not add a lot to our findings.

As mentioned earlier, one effective method to measure the learning effect of the learning activity is the use of an experimental group and a control group, such that the differences between these two groups can be objectively shown. The reason why we could not use this method was the fact that we could not split the group of students in two, simply for ethical reasons and because the learning activity took place in an official education institute. However, an idea was also to compare the results of this year to the results of last year, in which the students followed the previous lecture. We considered this, but decided that there were already too many differences between these two groups, so the observations between these two groups would not be representative. The main cause for this was that the previous year was still dealing with the Covid measures, so the given lectures and the education setting were significantly different compared to this year, that we could not use the previous year as a control group.

One method that we would recommend to other researches would be to look at more screen recordings of the students. In hindsight, these screen recordings provided a lot of insight into the students' coding process which we did not see anywhere else in the data sources. Especially if the voices of the students are also recorded, this could be a valuable data source. One drawback is that analysing these screen recordings take a lot of time, but in the end, we think that the useful observations of these recordings outweigh this drawback.

8.4 Looking back on the literature

In this section, we reflect back on the findings in the literature discussed in the Theoretical Framework and how our findings are related to these.

Teaching Design Patterns

In the literature, several problems were stated related to teaching design patterns for which no concrete solutions were found yet. In our research, we added several elements to our lecture which solved these problems, looking at our results.

Firstly, the abstract nature of the design patterns is difficult for the students. Even though the abstract UML of the design patterns has to be presented at one point, it helps to start with a concrete application of the pattern and afterwards, relate the abstract UML to a concrete UML. Furthermore, presenting some examples of the application of the pattern in an online quiz is also beneficial to tackle this abstractness.

Secondly, often static explanations are used for design patterns, while these patterns are in itself a very dynamic concept. Our first explanation is still of the static form, i.e. discussing the pattern in the slides, but afterwards, we present an implementation of the design pattern in a live coding session, which offered a dynamic explanation. For example, we used the debugger to show the students the flow of the program and how the different elements are related, which was beneficial for their understanding.

Thirdly, the utility of design patterns is often not clear to the students. This problem was mainly solved by the introductory program that was implemented without the design pattern. Afterwards, we saw that several problems occurred, which by solved one by one, which eventually lead to the discussed design pattern, which showed the benefits of using the pattern to the students. Furthermore, the examples in the online quiz also served as more examples where the usage of the pattern was beneficial.

Fourthly, it is unclear for the students when they should apply the pattern and when definitely not. This was solved by the addition of a refactoring exercise in the assignment and the examples in the online quiz as well, where the students had to indicate if and how design patterns could be applied in the example program.

Lastly, the literature already proposed that the usage of UMLs in design patterns is beneficial, but recurring problems here were that the UMLs for design patterns were very big, so students got easily overwhelmed, and students often had not enough experience and confidence with reading and using UMLs. The former point was solved by discussing the UML in detail, such that the students could follow along. The latter point was tackled by the introductory exercises related to UMLs, which had the purpose that all the students were on the same line at the start of the lecture. Moreover, the different exercises in the online quiz related to UMLs later in the lecture also served as exercises for students to get more familiar with UMLs.

Teaching Object-oriented Programming

Related to teaching Object-oriented Programming, our results were in line with the findings in the literature. Student struggle with the abstract nature of Object-oriented programming, but using a clear structure in the lecture with the usage of concrete examples tackles this problem.

The fact that students focus on the functionality of the program instead of the structure is solved by using an online quiz, where there are enough examples shown where an incorrect structure leads to problems. Furthermore, the complete lecture was focused on the structure of a program instead of the functionality, starting with the introductory exercises related to the different structures of a program.

The problem that students do not have enough experience yet to fully understand the benefits of design patterns is unfortunately very difficult to solve. However, the vast number of programs presented in this lecture could partially solve this problem, such that students have experienced more programs and potential problems.

The suggestion in the literature that the assignments should have clear criteria is a point on which we should improve. Currently, we expected that the assignment was clear enough for the students, but we got a lot of feedback that the assignment was unclear and value, which resulted in an assignment that was too difficult in general. Especially in the refactoring exercise was a lot of confusion, as students had never made such an exercise before, so the goal was quite unclear and which code they were able to change and what not.

Teaching Programming

Regarding teaching programming in general, the elements we put in our lecture had the same results as in the literature. A clear structure with a gradual build-up and easy examples helps the students to understand the material and to not be overwhelmed. The live coding sessions are useful as well, as the students find it easier to focus and it serves as a good bridge from theory to practice. Combining this with an online quiz with exercises, results in a interactive lecture in which students can easily focus, which were in line with the observations in the literature.

8.5 Future work

In this section, we recommend changes for the designed learning activity, which include the lecture, the assignment and a possible Tutorial session, which could be an extra session in the week.

Lecture

The would propose the following changes in the lecture:

- Put emphasis on function bodies in design patterns
  To put more emphasis on the function bodies in the design patterns, we suggest to present this code next to the UML, such that the students

see that these function bodies are also "part" of the whole structure. Currently, the function bodies are only shown on a few slides, so we think that the importance was not emphasised enough.

- Add a slide for the function calls to use the design patterns
  One slide could be added where the calls to use the design patterns are explicitly discussed. Currently, these calls are only mentioned a few time in the slides and in the data, this was one of the most occurring problems.

- Emphasize the difference and relation between the `accept` and `visit` function
  Specifically for the Visitor pattern, it would be wise to explicitly discuss the difference between the `accept` and `visit` functions. Not only was this not clear to all students, but this is also a useful insight into the typings of different functions.

- Change the example program of the Strategy Pattern
  The current example used for the Strategy Pattern was a routing algorithm. However, we experienced that the code used for this program was relatively difficult, as students did not learned about routing algorithms yet, thus the code was unnecessarily complicated.

- Put the code in the live coding session in the slides
  One could try to put the code discussed in the live coding sessions in the slides as well. These slides are not discussed, as the code is already presented in the coding session, but the slides are still complete. This is relatively little work for the teacher, as the code is already written in a program, so it only has to be copied. Furthermore, the mixed opinions on the live coding sessions are also satisfied with this addition.

- Cut some material in the slides
  We experienced in our lecture that we did not have enough time to discuss the complete slide set; the last slides about design patterns in general was skipped. Therefore, the teacher could remove some slides or topics in the lecture, but unfortunately, we could not think of any obvious cuts. One could consider to add an extra lecture or move some topics to the extra tutorial session.

- Make a slide set with exercises and without
  Related to publishing the slides afterwards, it is wise to publish a slide set which contains the exercises and one which does not. We got feedback from students that the slides were cluttered with exercises, so with these 2 versions, the students can decide which version they want to use.

It is also interesting to see how the students experience these lectures when the structure is used for several weeks. Currently, the usage of the online quiz and the live codings sessions were a change of pace for students and felt refreshing, which led to positive feedback. However, when this structure is applied for several weeks, it is interesting to see if students still experience these advantages or if they, for example, do not participate in the online quiz anymore.

Assignment

For the two-parted assignment, we have recommendations for part 1 (refactoring exercise), part 2 (implementation exercise) and general comments:

Part 1

- State which files the students can change
  State explicitly that the students can change all files, except for the test cases. Currently, the students were confused which files they could change and which ones were not supposed to be changed.

- Elaborate on the encryption methods
  We received some feedback that the encryption methods were difficult to understand, so these could be elaborated in more detail. However, this feedback was only provided by a small number of students and we think that the current explanation is long enough.

Part 2

- Remove the usage of the Standard Library and the enumeration for the Binary Operator
  The current task of the Binary Operator was too difficult for students, mainly because it was a combination of an enumeration, importation of a Standard Library and a Strategy pattern. We have two solutions for this:

  1. Completely remove the Binary Operator interface and replace this by the concrete classes And, Or and Implies. This solves all problems and it probably the most intuitive for students.
  2. Remove the usage of the Standard Library and enumeration, but keep the Strategy pattern. So the Binary Operator class is still kept, but it uses an Operator interface, which are implemented by And, Or and Implies and are added as an attribute to Binary Operator. If the assignment provides some hints on how to do this, students still use a Strategy Pattern, but without the difficulties arising from the Standard Library and enumeration. This

implementation is hopefully a lot more intuitive, and will save the problems a lot of time.

- Provide proper support for the printing of brackets in the ShowVisitor
Another problem was the placements of brackets in the ShowVisitor where the precedence of the different operators are important. Students had a lot of difficulties with this precedence function and how this exactly worked. Therefore, an explanation about this precedence and how this exactly works is necessary for the students to make this assignment, instead of a comment that the students have to figure out the details themselves.

- State what the printed result of the ShowVisitor should be
A small addition would be to add some examples of the expected printing of the ShowVisitor, for example "=>" is used for implies and "\/" for or. Students can see this in the test cases, but it would not hurt to state this in in the assignment.

- Remove the sequence diagram
We would suggest to remove the sequence diagram at the end of this exercise. We understand its purpose, but in practice, students do not understand how a sequence diagram works, so this exercise leads to more problems than it contributes to the learning goals.

General remarks

- Split the assignment in more sections
We got feedback from the students that they got overwhelmed by the current assignment and one solution could be to split the assignment into more sections. For example, the current section "Logical formulas" in part 2 could be split up further in "Logical formulas", "Visitor pattern" and an additional section for the Strategy pattern in Binary Operator.

- Provide concrete tasks with which the students should start
Students also found it hard to start with the assignment, so it would help to give some specific instructions on how to start. Note that the list of instructions provided in the assignment for part 2 was already very useful for students. But an addition could be to start with the classes Not, Constant and Atom, afterwards, do And, Or, Implies with the Binary Operator etc. With these instructions, students can start with relative easy tasks and once they are programming, the other tasks follow more easily.

Possible tutorial session

Even though the tutorial session was outside the scope of our research, we got some feedback on what the students would like to see in an extra session between the lecture and the practical sessions for the assignment. The feedback here is discussed with some ideas for such a session:

- Discuss the assignment with students
  Students mentioned that they struggle with understanding the assignment, so they would find it useful if the assignment is discussed with the students in this session. So, the lecture does not immediately start with presenting code snippets used for the assignment, but first discussing the whole assignment with the students and tell them what is expected from them.

- Provide hints for difficult details in the assignment
  Currently, the tutorial sessions mostly presented and discussed code for the assignment, which was experienced as quite useful by the students. For example, in this week, that could be discussing the usage of the Strategy pattern inside the Binary Operator or the usage of precedence inside the ShowVisitor. Note that with discussing the assignment beforehand, students can actually place the presented code snippets in context of the whole assignment.

- Let the students code at the tutorial
  The students would like some more interaction in the tutorial session and one solution could be to let the students code themselves in this session. For example, the teacher could ask the students to implement one specific part of the program and afterwards, write the necessary code themselves live. Not only turns this session into an active learning session, but students also have some starting code at the end of the tutorial. Moreover, the students get the benefits of a live coding session in the tutorial as well.

- Refactoring example
  Specifically for design patterns, it would be wise to do an example related to refactoring a program and how to tackle this as students. Currently, this is the first time that students are asked to refactor a program themselves, so many were did not know what was expected from them and were they could start. Providing support for this would be quite useful.

References

Agbo, F. J., Oyelere, S. S., Suhonen, J., & Adewumi, S. (2019). A systematic review of computational thinking approach for programming education in higher education institutions. In *Proceedings of the 19th koli calling international conference on computing education research.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3364510.3364521 doi: 10.1145/3364510.3364521

Alaagib, N. (2019). Comparison of the effectiveness of lectures based on problems and traditional lectures in physiology teaching in sudan.. Retrieved from https://bmcmededuc.biomedcentral.com/articles/10.1186/s12909-019-1799-0 doi: https://doi.org/10.1186/s12909-019-1799-0

Alammary, A. (2019). Blended learning models for introductory programming courses: A systematic review. In *Plos one* (Vol. 14-9). Retrieved from https://doi.org/10.1371/journal.pone.0221765

Anggrawan, A., Ibrahim, N., Suyitno, P., & Satria, C. (2018, 10). Influence of blended learning on learning result of algorithm and programming. In (p. 1-6). doi: 10.1109/IAC.2018.8780420

Azimullah, Z., An, Y. S., & Denny, P. (2020). Evaluating an interactive tool for teaching design patterns. In *Proceedings of the twenty-second australasian computing education conference* (p. 167–176). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3373165.3373184 doi: 10.1145/3373165.3373184

Brown, N. C. C., & Wilson, G. (2018, 04). Ten quick tips for teaching programming. *PLOS Computational Biology*, *14*(4), 1-8. Retrieved from https://doi.org/10.1371/journal.pcbi.1006023 doi: 10.1371/journal.pcbi.1006023

Chibizova, N. (2018). The problems of programming teaching. In *2018 iv international conference on information technologies in engineering education (inforino)* (p. 1-4). doi: 10.1109/INFORINO.2018.8581834

Chis, A., Moldovan, A.-N., Murphy, L., & Muntean, C. (2018, 01). Investigating flipped classroom and problem-based learning in a programming module for computing conversion course. *Educational Technology and Society*, *21*, 232-247.

Eppley, K., & Dudley-Marling, C. (2018, 06). Does direct instruction work?: A critical assessment of direct instruction research and its theoretical perspective. *Journal of Curriculum and Pedagogy*, *16*, 1-20. doi: 10.1080/15505170.2018.1438321

Gutierrez, L., Guerrero, C., & López-Ospina, H. (2022, 06). Ranking of problems and solutions in the teaching and learning of object-oriented programming. *Education and Information Technologies*, *27*, 1-35. doi: 10.1007/s10639-022-10929-5

Hayashi, Y., Fukamachi, K.-I., & Komatsugawa, H. (2015). Collaborative learning in computer programming courses that adopted the flipped classroom. In *2015 international conference on learning and teaching in computing and engineering* (p. 209-212). doi: 10.1109/LaTiCE.2015.43

Hlescu, A. A., Birlescu, A. E., Hanganu, B., Manoilescu, I. S., & Ioan, B. G. (2020, Sep.). Traditional teaching versus online teaching of forensic autopsy case study – grigore t. popa university of medicine and pharmacy in iasi. *Revista Romaneasca pentru Educatie Multidimensionala*, *12*(2Sup1), 41-54. Retrieved from `https://www.lumenpublishing.com/journals/index.php/rrem/article/view/2831` doi: 10.18662/rrem/12.2Sup1/288

Keung, J., Xiao, Y., Mi, Q., & Lee, V. C. S. (2018). Bluej-uml: Learning object-oriented programming paradigm using interactive programming environment. In *2018 international symposium on educational technology (iset)* (p. 47-51). doi: 10.1109/ISET.2018.00020

Khakim, A. A. (2019/08). Problem-based learning in programming lesson. In *Proceedings of the 2nd international conference on intervention and applied psychology (iciap 2018)* (p. 529-536). Atlantis Press. Retrieved from `https://doi.org/10.2991/iciap-18.2019.44` doi: https://doi.org/10.2991/iciap-18.2019.44

Lotlikar, P., & Kussmaul, C. (2022). Guiding students to learn about design patterns with process oriented guided inquiry learning (pogil). In *Proceedings of the 27th conference on pattern languages of programs.* USA: The Hillside Group.

Lytle, N., Catete, V., Isvik, A., Boulden, D., Dong, Y., Wiebe, E., & Barnes, T. (2019, 10). From 'use' to 'choose': Scaffolding ct curricula and exploring student choices while programming (practical report). In (p. 1-6). doi: 10.1145/3361721.3362110

Majherová, J., & Králík, V. (2017, 09). Innovative methods in teaching programming for future informatics teachers. *European Journal of Contemporary Education*, *2017*, 390-400. doi: 10.13187/ejced.2017.3.390

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2019). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, *62*(2), 77-90.

doi: 10.1109/TE.2018.2864133

Mello Fonseca, F., Bezerra da Silva, E., & Silveira Mendonça, D. (2019). Designing dojo: A collaborative method for teaching design patterns. In *2019 ieee/acm 12th international workshop on cooperative and human aspects of software engineering (chase)* (p. 39-40). doi: 10.1109/CHASE.2019.00017

Qian, Y., Hambrusch, S., Yadav, A., Gretter, S., & Li, Y. (2019, 04). Teachers' perceptions of student misconceptions in introductory programming. *Journal of Educational Computing Research*, *58*, 073563311984541. doi: 10.1177/0735633119845413

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137-172. Retrieved from https://doi.org/10.1076/csed.13.2.137.14200 doi: 10.1076/csed.13.2.137.14200

Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th acm technical symposium on computer science education* (p. 651–656). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2445196.2445388 doi: 10.1145/2445196.2445388

Santos, S. C., Tedesco, P. A., Borba, M., & Brito, M. (2020). Innovative approaches in teaching programming: A systematic literature review. In *Proceedings of the 12th international conference on computer supported education* (pp. 205–2014).

Sharma, M., Biros, D., Ayyalasomayajula, S., & Dalal, N. (2020a). Teaching programming to the post-millennial generation: Pedagogica considerations for an is course. In *Journal of information systems education* (Vol. 31, p. 96-105). Retrieved from https://aisel.aisnet.org/jise/vol31/iss2/2

Sharma, M., Biros, D., Ayyalasomayajula, S., & Dalal, N. (2020b, 06). Teaching programming to the post-millennial generation: Pedagogic considerations for an is course. *Journal of Information Systems Education*, *31*, 96-105.

Shosse, K. (2022). *What's a design pattern?* Retrieved from https://refactoring.guru/design-patterns/what-is-pattern

Silva, D., Schots, M., & Duboc, L. (2019, 10). Fostering design patterns education: An exemplar inspired in the angry birds game franchise. In (p. 168-177). doi: 10.1145/3364641.3364660

Stamouli, I., & Huggard, M. (2006). Object oriented programming and program correctness: The students' perspective. In *Proceedings of the second international workshop on computing education research* (p. 109–118). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1151588.1151605 doi: 10.1145/1151588.1151605

Stuurman, S., Passier, H., & Barendsen, E. (2016, November 24). Analyzing student's software redesign strategies. In J. Sheard & C. Suero Montero (Eds.), *Koli calling '16: Proceedings of the 16th koli calling international conference on computing education research* (pp. 110–119). United States: Association for Computing Machinery (ACM). (16th Koli Calling International Conference on Computing Education Research, Koli Calling '16 ; Conference date: 24-11-2016 Through 27-11-2016) doi: 10.1145/2999541.2999559

Sun, Q., Wu, J., & Liu, K. (2020). Toward understanding students' learning performance in an object-oriented programming course: The perspective of program quality. *IEEE Access*, *8*, 37505-37517. doi: 10.1109/ACCESS.2020.2973470

Tillmann, N., Moskal, M., de Halleux, J., Fahndrich, M., Bishop, J., Samuel, A., & Xie, T. (2012). The future of teaching programming is on mobile devices. In *Proceedings of the 17th acm annual conference on innovation and technology in computer science education* (p. 156–161). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi-org.ru.idm.oclc.org/10.1145/2325296.2325336` doi: 10.1145/2325296.2325336

Weiss, C. (2020, 2-4 March, 2020). Small group learning in larger lecture classes. In *Inted2020 proceedings* (p. 7154-7161). IATED. Retrieved from `https://dx.doi.org/10.21125/inted.2020.1892` doi: 10.21125/inted.2020.1892

Xinogalos, S. (2015, jul). Object-oriented design and programming: An investigation of novices' conceptions on objects and classes. *ACM Trans. Comput. Educ.*, *15*(3). Retrieved from `https://doi.org/10.1145/2700519` doi: 10.1145/2700519

Yu, X., Yang, Y., & Wu, X. (2021). Exploration and practice of object-oriented programming in the mode of "internet +" education. In *2021 2nd international conference on computers, information processing and advanced education* (p. 709–711). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3456887.3457047` doi: 10.1145/3456887.3457047

Appendix A

Slides lecture

Design Patterns

Object Oriented Programming week 10

Ruben Holubek

Who am I?

- Ruben Holubek
- 2nd year Master Student Software Science
- Research related to teaching programming
- Right now: how to effectively teach OO programming, especially design patterns

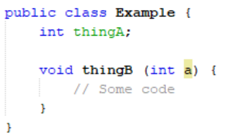
Online Quiz

- Please join the online quiz for exercises:
- Please go to www.socrative.com
- Click on login (upper right corner)
- Click on Student login
- Insert room name HOLUBEK60

Question 1

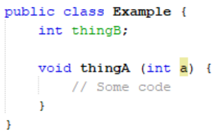
Which code fragment is represented by this UML?

A.



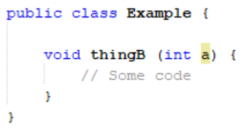
```
public class Example {
    int thingA;

    void thingB (int a) {
        // Some code
    }
}
```

B.



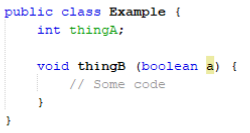
```
public class Example {
    int thingB;

    void thingA (int a) {
        // Some code
    }
}
```

C.

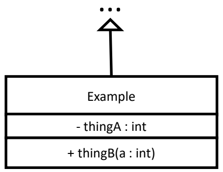


```
public class Example {

    void thingB (int a) {
        // Some code
    }
}
```

D.



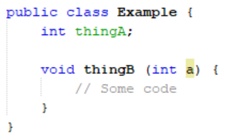
```
public class Example {
    int thingA;

    void thingB (boolean a) {
        // Some code
    }
}
```



Question 1

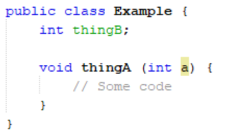
A is correct; B switched around the names for the attribute and function, the attribute thingA is missing in C and the type for thingB is incorrect in D

A.



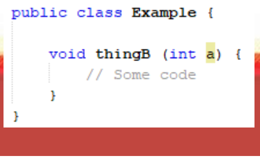
```
public class Example {
    int thingA;

    void thingB (int a) {
        // Some code
    }
}
```

B.



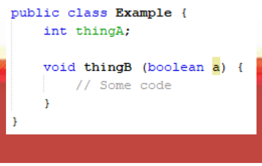
```
public class Example {
    int thingB;

    void thingA (int a) {
        // Some code
    }
}
```

C.

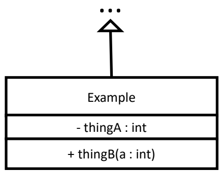


```
public class Example {

    void thingB (int a) {
        // Some code
    }
}
```

D.

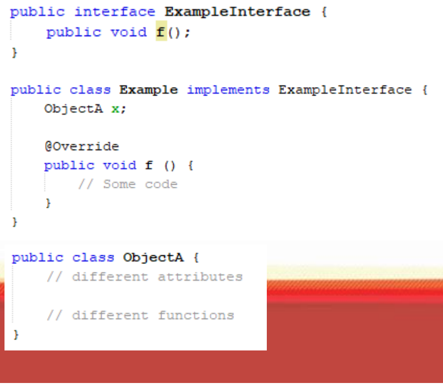


```
public class Example {
    int thingA;

    void thingB (boolean a) {
        // Some code
    }
}
```



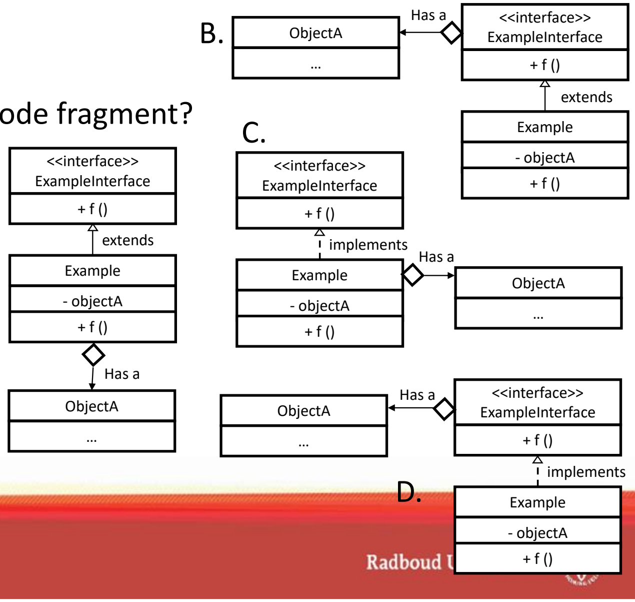
Question 2

Which UML represents this code fragment?



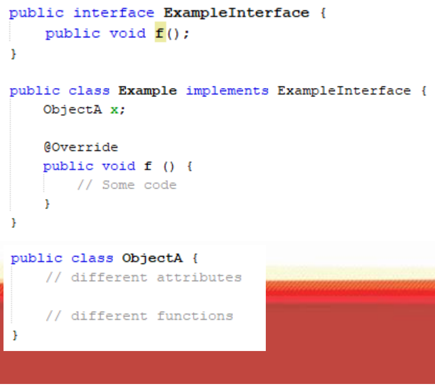
```
public interface ExampleInterface {
    public void f();
}

public class Example implements ExampleInterface {
    ObjectA x;

    @Override
    public void f () {
        // Some code
    }
}

public class ObjectA {
    // different attributes

    // different functions
}
```

A.



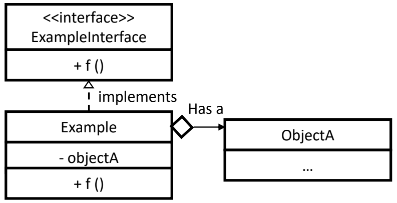
Question 2

C is correct, as Example implements ExampleInterface (it doesn't extend it) and Example uses ObjectA (ExampleInterface does not)



```
public interface ExampleInterface {
    public void f();
}

public class Example implements ExampleInterface {
    ObjectA x;

    @Override
    public void f () {
        // Some code
    }
}

public class ObjectA {
    // different attributes

    // different functions
}
```

C.



Question 3

Which of the following programs have a similar structure? (e.g. containing a loop, using similar interfaces etc)

1. Getting the last names of a list of students
2. Evaluating the value of a given boolean formula (e.g. T /\ F -> T)
3. For the numbers between 1 and 100, calculate the square root

A. 1 and 2 have a similar structure

B. 1 and 3 have a similar structure

C. 2 and 3 have a similar structure

D. None of these programs have a similar structure

Question 3

1. Getting all the last names of a list of students
2. Evaluating the value of a given boolean formula (e.g. T /\ F -> T)
3. For the numbers between 1 and 100, calculate the square root

A. 1 and 2 have a similar structure

B. 1 and 3 have a similar structure

C. 2 and 3 have a similar structure

D. None of these programs have a similar structure

Both of these formulas iterate over something (list of students or numbers) and perform an action on each element (return the name or calculate square root). Program 2 doesn't perform such an iterating operation.

Question 4

Which of the following objects have a similar structure?

1. Many different bikes with an option to add a specific, special saddle

2. A pizza with many options for different toppings, e.g. tomato, cheese, chicken etc

3. A server where users can have a combination of different rights, e.g. reading, writing, editing, adding new users etc

1. 1 and 2 have a similar structure

A. 1 and 3 have a similar structure

B. 2 and 3 have a similar structure

C. None of these programs have a similar structure

Question 4

1. Many different bikes with an option to add a specific, special saddle

2. A pizza with many options for different toppings, e.g. tomato, cheese, chicken etc

3. A server where users can have a combination of different rights, e.g. reading, writing, editing, adding new users etc

A. 1 and 2 have a similar structure

B. 1 and 3 have a similar structure

C. 2 and 3 have a similar structure

D. None of these programs have a similar structure

Both these programs have one base object (pizza or user), which can have multiple "extras" (toppings or rights). Program 1 has different base objects (bikes), but with only one "extra" (special saddle), so this is a simple attribute

Design patterns

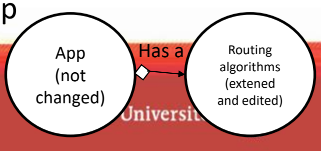
Design Patterns

- Design patterns are specific structures that reoccur in different programs
  - Iterating over a loop and performing a specific action
  - Having a base object with many different possibilities of extras
  - And many more…
- You probably used a lot of these without even thinking about it
- Now we simply give them a name
- This lecture: we discuss 3 different design patterns
  - Strategy pattern
  - Decorator pattern
  - Visitor pattern

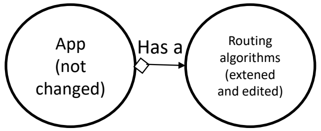
Strategy Pattern

Problem

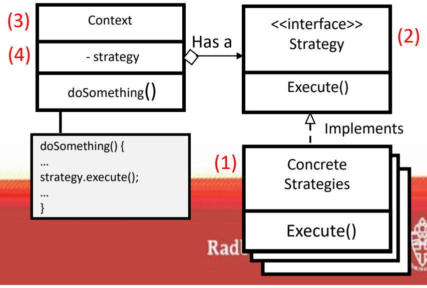
- Navigation app that provides the fastest route for 2 given points
- First version: provides routes traveling by car
- Later versions: algorithms for other routes
  - Biking, walking, busses, along highlights, shortest route etc.
- Maintaining this without a proper structure is impossible
  - Code gets messy
  - Altering the code for the whole navigation app, while only changing the routing algorithms
- Idea: Split the routing algorithms from the whole app
- Solution: Strategy Pattern



Strategy Pattern

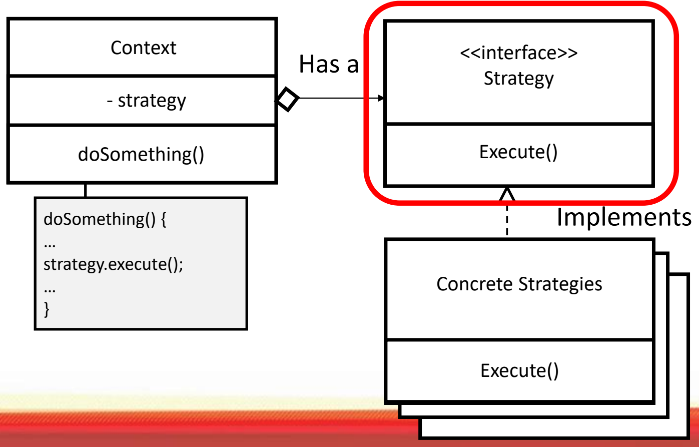


- Previously: Split the routing algorithms from the whole app
- Strategy Pattern: Split the strategies from the context
- Applied with “a class that does something specific in a lot of ways”
- The concrete strategies (1) are all “stored” in the interface Strategy (2)
- The context (3) has a Strategy attribute (4)
- The context uses this reference as a “generic interface”
- The actual strategies are completely split from the context!



```
doSomething() {
...
strategy.execute();
...
}
```

Strategy Pattern UML

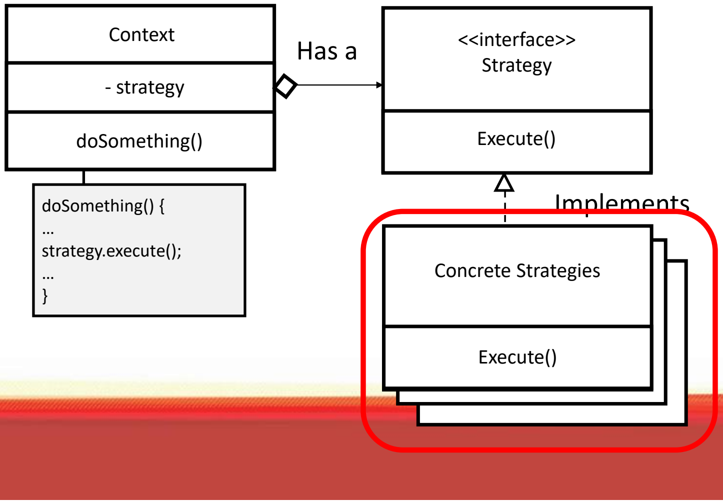


```
doSomething() {
…
strategy.execute();
…
}
```

Interface Strategy

- The interface for the implemented strategies
- It has an abstract function execute() which every concrete strategy should implement

Strategy Pattern UML

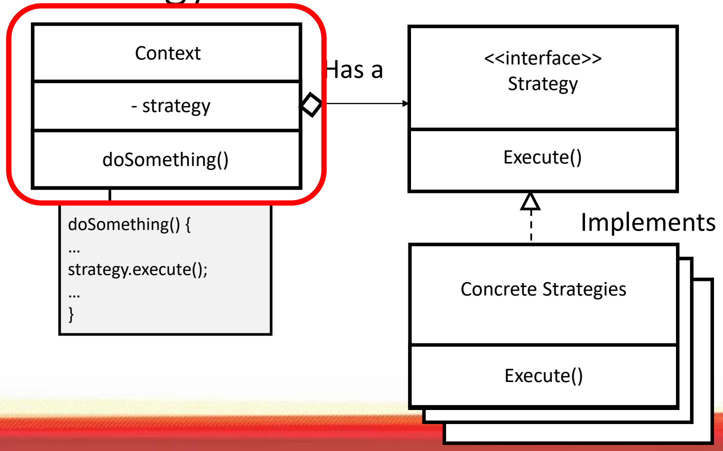


```
doSomething() {
…
strategy.execute();
…
}
```

The concrete strategies

- These implement the Strategy interface
- So every strategy has an implemented Execute function

Strategy Pattern UML

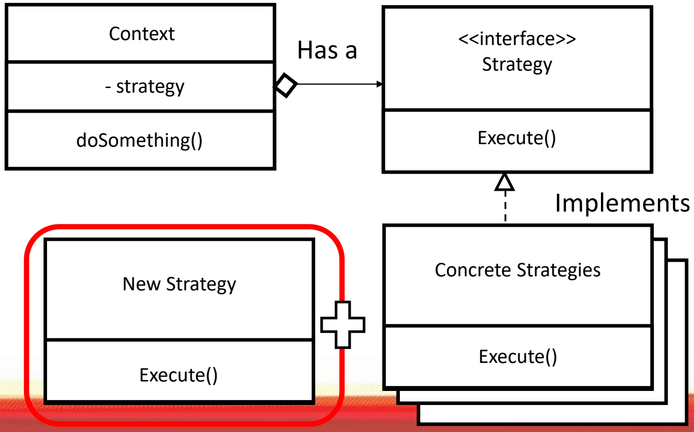


```
doSomething() {
…
strategy.execute();
…
}
```

Context

- The context has a reference to the used strategy
- In the functionality of Context, there is a function doSomething() that uses the stored strategy
- E.g. strategy.execute()
- Context is implemented with an abstract strategy!

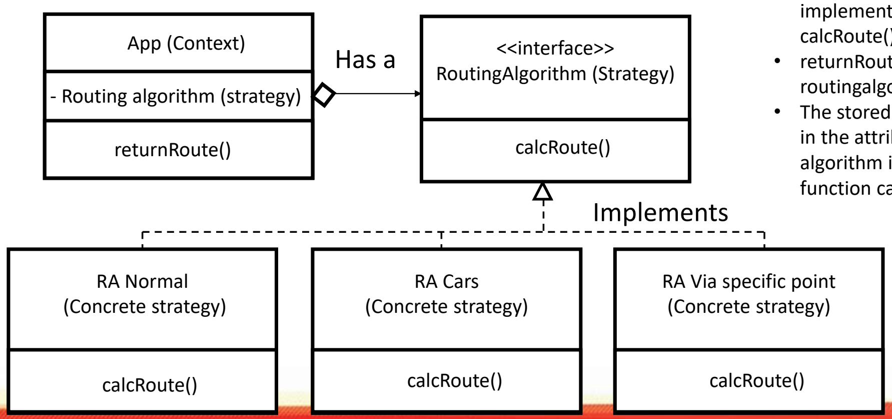
Strategy Pattern UML



Adding a new strategy

- Implement the Execute function for this strategy
- Done!
- Context is unchanged!
- Context works with an abstract strategy, so as long as a strategy has an execute function, it can use it (which it automatically has as it implements the interface strategy!)

Application on previous problem



App (Context)

- Routing algorithm (strategy)

returnRoute()

Has a

<<interface>>
RoutingAlgorithm (Strategy)

calcRoute()

Implements

RA Normal
(Concrete strategy)

calcRoute()

RA Cars
(Concrete strategy)

calcRoute()

RA Via specific point
(Concrete strategy)

calcRoute()

- All concrete routing algorithms implement the interface and have an implementation of calcRoute()
- returnRoute() uses routingalgorithm.calcRoute()
- The stored RoutingAlgorithm in the attribute decides which algorithm is used with this function call!

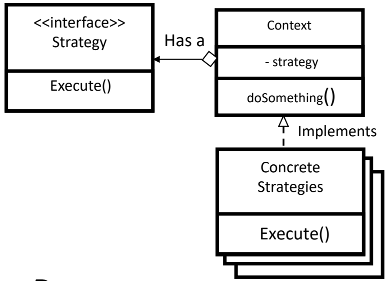
Live Coding Session

- Lets take a look at the program from the previous example
- This can be found in StrategyPatternExample.zip
- Take a look at it in your own time as well and experiment!
- Hint: You can also use the debugger to investigate the flow of the program

Question 5

Which is a correct UML for the strategy pattern?

C.



<<interface>>
Strategy

Execute()

Has a

Context

- strategy

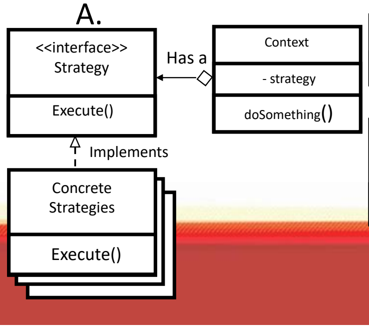
doSomething()

Implements

Concrete Strategies

Execute()

A.



<<interface>>
Strategy

Execute()

Has a

Context

- strategy

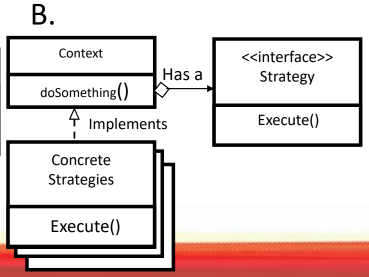
doSomething()

Implements

Concrete Strategies

Execute()

B.



Context

doSomething()

Has a

<<interface>>
Strategy

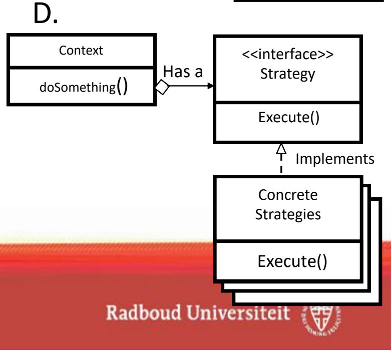
Execute()

Implements

Concrete Strategies

Execute()

D.



Context

doSomething()

Has a

<<interface>>
Strategy

Execute()

Implements

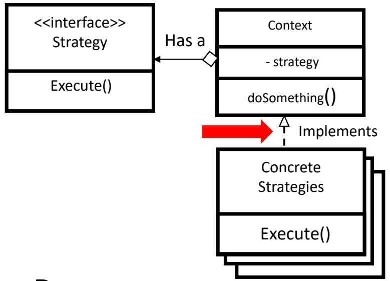
Concrete Strategies

Execute()

Question 5

A is correct. The Concrete Strategies should Implement the interface Strategy (so A and D remain). Furthermore, the Context should have an attribute with the used strategy, so A is correct

C.



<<interface>>
Strategy

Execute()

Has a

Context

- strategy

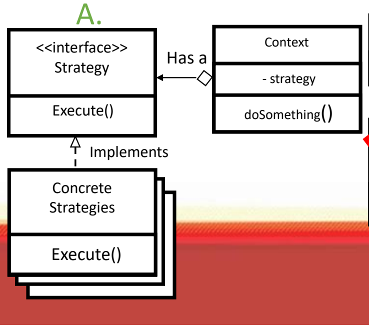
doSomething()

Implements

Concrete Strategies

Execute()

A.



<<interface>>
Strategy

Execute()

Has a

Context

- strategy

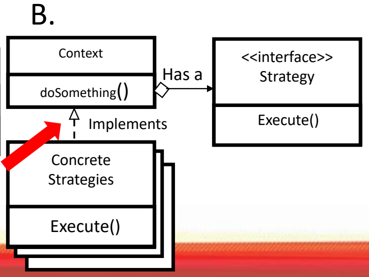
doSomething()

Implements

Concrete Strategies

Execute()

B.



Context

doSomething()

Has a

<<interface>>
Strategy

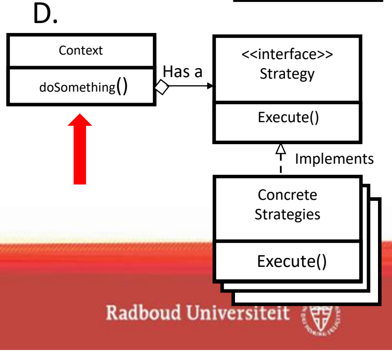
Execute()

Implements

Concrete Strategies

Execute()

D.



Context

doSomething()

Has a

<<interface>>
Strategy

Execute()

Implements

Concrete Strategies

Execute()

Question 6

Suppose the following program:

We have a program that encrypts a given message with either RSA or Triple DES (2 encryption methods). The program probably won't be extended with other encryption methods.

Should we use the strategy pattern and how?

A. Yes, the encryption methods are the strategies

B. Yes, the different messages are the strategies

C. No, it is not necessary to apply the pattern

Question 6

We have a program that encrypts a given message with either RSA or Triple DES (2 encryption methods). The program probably won't be extended with other encryption methods.

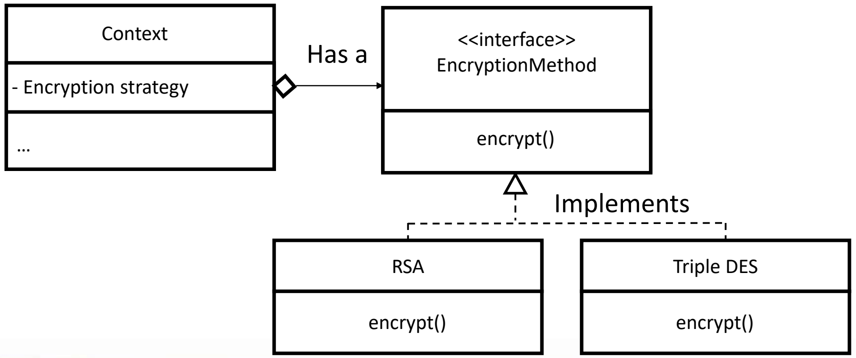
A. Yes, the encryption methods are the strategies

B. Yes, the different messages are the strategies

C. No, it is not necessary to apply the pattern

These encryption methods can be applied as strategies. However, it is debatable whether that is better, as there are only 2 methods used. Therefore, both answers are correct, but the preference is to still apply the strategy pattern for future adjustments

Question 6



Context

- Encryption strategy

…

Has a

<<interface>>
EncryptionMethod

encrypt()

Implements

RSA

encrypt()

Triple DES

encrypt()

Question 7

Suppose the following program:

We have a simulation of robots which move in a field. These robots have different behaviors, e.g. defensive, offensive or random. These behaviors can also be combined with each other or can have different variations.

Should we use the strategy pattern and how?

A. Yes, the robots are the strategies

B. Yes, the behaviors are the strategies

C. No, it is not necessary to apply the pattern

Question 7

We have a simulation of robots which move in a field. These robots have different behaviors, e.g. defensive, offensive or random. These behaviors can also be combined with each other or can have different variations.

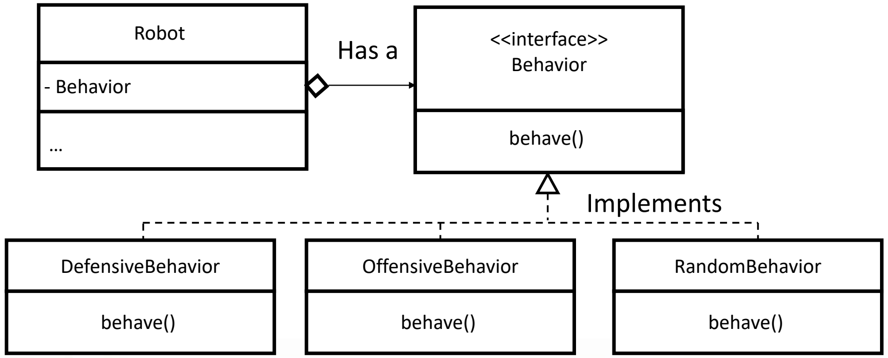
A. Yes, the robots are the strategies

B. Yes, the behaviors are the strategies

C. No, it is not necessary to apply the pattern

The behaviors are the strategies used by the robots; see the UML on the next slide

Question 7



Robot

- Behavior

...

Has a

<<interface>>
Behavior

behave()

Implements

DefensiveBehavior

behave()

OffensiveBehavior

behave()

RandomBehavior

behave()

Question 8

Suppose the following program:

We have a program that extracts plain text from different types of files, e.g. pdf, xml or html. This is then returned as a string.

Should we use the strategy pattern and how?

A. Yes, the different types of files are the strategies

B. Yes, the different extraction algorithms are the strategies

C. No, it is not necessary to apply the pattern

Question 8

We have a program that extracts plain text from different types of files, e.g. pdf, xml or html. This is then returned as a string.

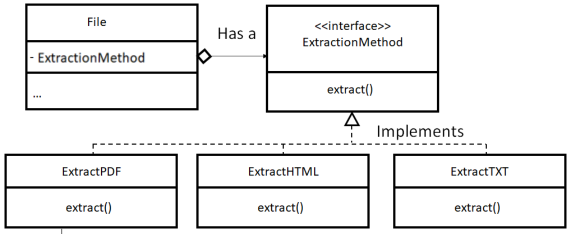
A. Yes, the different types of files are the strategies

B. Yes, the different extraction algorithms are the strategies

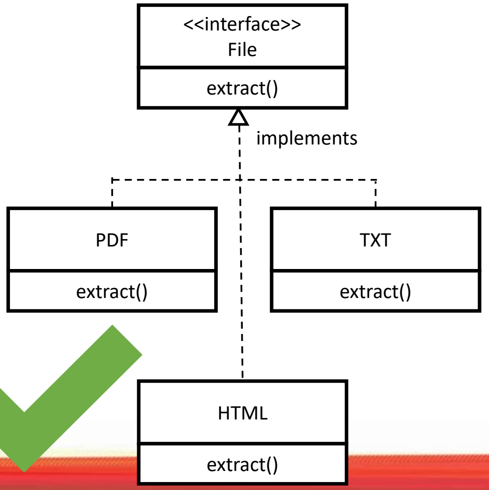
C. No, it is not necessary to apply the pattern

Using the Strategy pattern here will result in non-modular code, as the different methods for the specific text files can only be applied on the corresponding text file instead of universally on all files. It is better to have an abstract function extract() implemented for all the different files. See the next slide for the UMLs.

Question 8



```
extract() {
  if (file.getClass() == pdf.getClass()) {
  // do something
  } else {
    ???
  }
}
```

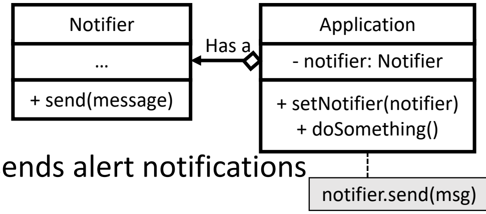


Recap

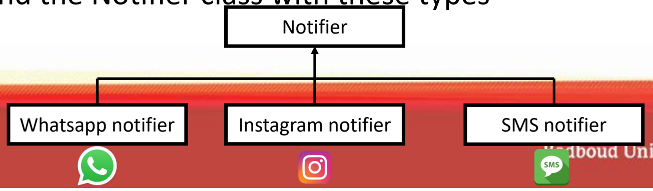
- The Strategy pattern can be used with classes that do something specific with different strategies
  - E.g. calculating a route, different encryption methods, different behaviors…
- It splits the different strategies (by using an interface) from the context
- The code in the context is unchanged if
  - A strategy contains a bug and is fixed
  - A new strategy is added

Decorator Pattern

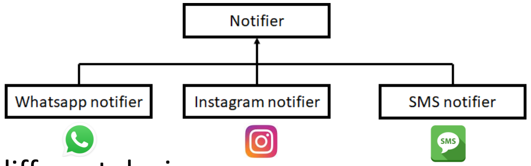
Problem I



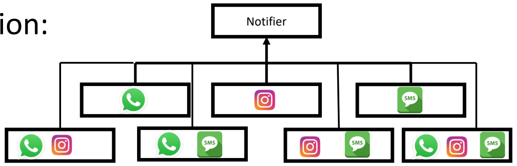
- Suppose we have an application which sends alert notifications
- First only email notifier
  - Solution: make a separate Notifier class
- Afterwards, demand for additional notifiers, next to the standard email notifier
  - E.g. Whatsapp, Instagram, SMS notification
- Solution: Extend the Notifier class with these types



Problem II



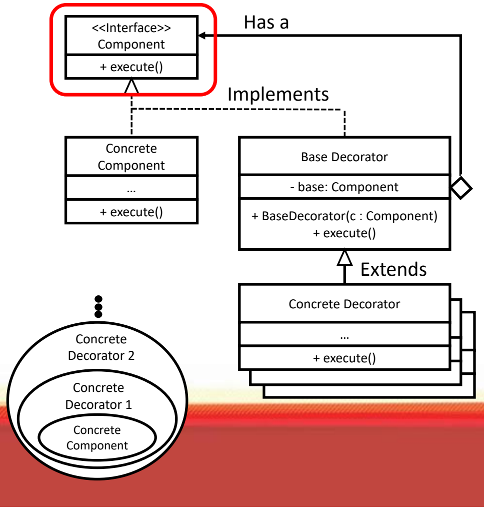
- Now: persons want notifiers on different devices
  - E.g. Whatsapp and SMS notifier
- “Solution”: Lets make subclasses for all these combinations:
- Unfortunately, this is not a great solution:
  - Duplicate code
  - A lot of unnecessary code
  - Not extendable
    - Imagine adding one extra Notifier
- Solution: **Decorator Pattern**



Decorator Pattern

- Previous problem: add multiple notifiers to the basic email notifier
- Decorator Pattern: add multiple decorators to the concrete component
  - E.g. multiple toppings on a pizza
- Instead of subclasses for all combinations
- Simply add the desired decorators to the concrete component
- Advantages:
  - No duplicate code
  - Easily extendable with new decorators
  - A lot of flexibility

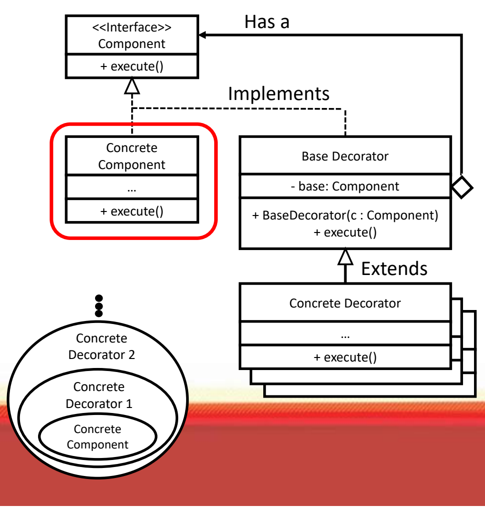
Decorator Pattern UML



Interface Component

- This is the interface for both the Concrete Component and the Decorators
- It contains a function which is implemented for all Concrete Components and Decorators
  - E.g. a cost() or toString() function for the pizzas with toppings

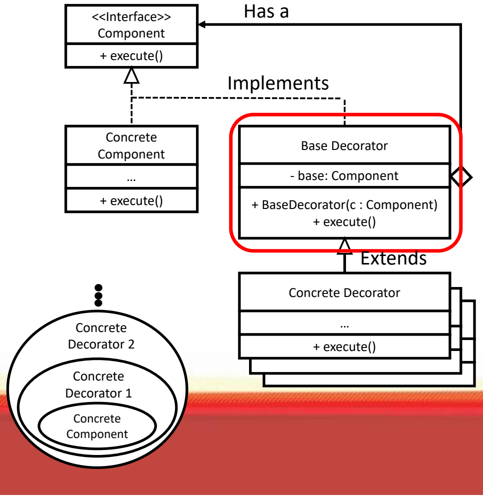
Decorator Pattern UML



Concrete Component

- This is the base on which the Decorators can be “wrapped”
  - E.g. the base of a pizza on which we can “wrap” the toppings
- Note that we have the same function execute() here as well
- Note that we **don’t** have a Component object in a Concrete Component, as it is a base and it cannot be wrapped around something in contrast to a Decorator

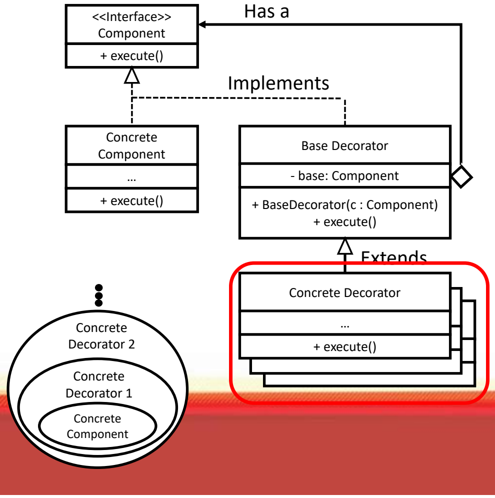
Decorator Pattern UML



Base Decorator

- This serves as the standard Decorator for all the Concrete decorators
- It contains a Component object, which is the object on which the current Decorator is wrapped
  - Note that this can either be a Concrete Component or a Concrete Component with several Decorators wrapped around it represented as a Component
- It contains a Constructor which simply sets the base
- It contains an execute() function, which is standard defined as base.execute() (simply calls the execute of the stored base)
- Note that the arrow with a diamond states that the Base Decorator uses a Component

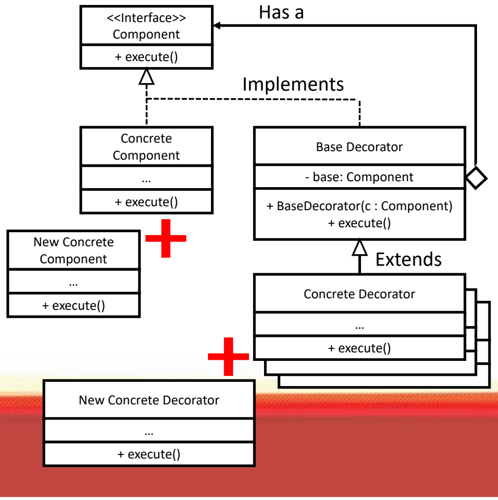
Decorator Pattern UML



Concrete Decorator

- These are the concrete Decorators that **extend** the Base Decorator
- It overwrites the execute() function of the Base Decorator with the extra functionality of the Concrete Decorator
- The overwritten execute() function contains a call super.execute(), which simply calls the execute() function of the extended object, in this case the Base Decorator, thus calling the execute() function of the stored base Component.
- The remainder of the overwritten execute() function is the addition of the Decorator
  - E.g. adding a topping of 35 cents to the rest of the pizza
  - cost() {return super.cost() + 0.35}

Decorator Pattern UML



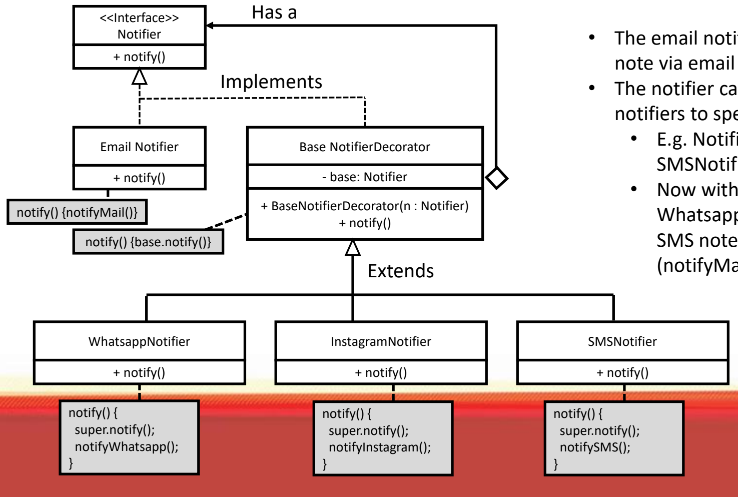
Adding new Concrete Component

- Only the execute() function of this new Component should be implemented
  - E.g. the cost of a different base for a pizza
- It can immediately be used in the whole system while not changed any code in any other class!

Adding new Concrete Decorator

- Similarly, only the execute() function of this new Decorator should be implemented
  - E.g the cost of a new topping for a pizza
- It can immediately be used in the whole system while not changed any code in any other class!

Application on the previous problem



```
notify() {
  super.notify();
  notifyWhatsapp();
}
```

```
notify() {
  super.notify();
  notifyInstagram();
}
```

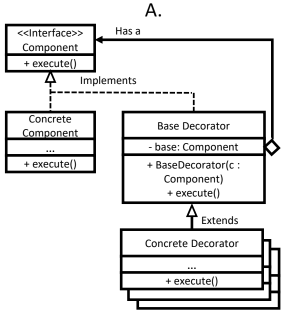
```
notify() {
  super.notify();
  notifySMS();
}
```

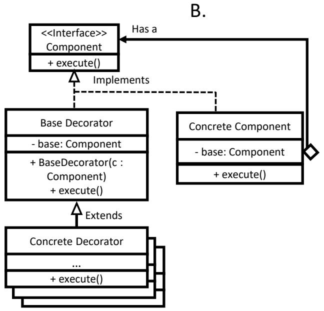
- The email notifier is the base, as everyone gets a note via email
- The notifier can then be wrapped with other notifiers to specify the desired notifiers
  - E.g. Notifier n = new WhatsappNotifier(new SMSNotifier(new EmailNotifier));
  - Now with n.notify, the user receives a Whatsap note (notifyWhatsapp() is called), a SMS note (notifySMS() is called) and an email (notifyMail() is called)

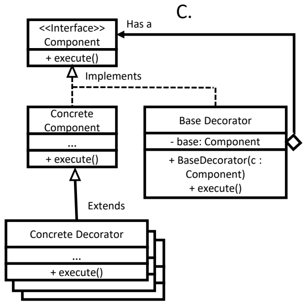
Live coding session

- Lets take a look at the program from the previous example
- This can be found in DecoratorPatternExample.zip
- Take a look at it in your own time as well and experiment!

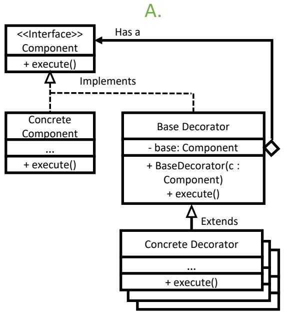
Question 9: Correct UML for Decorator Pattern?

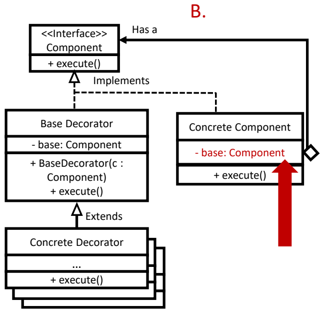


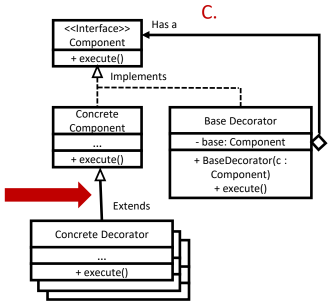




Question 9: Correct UML for Decorator Pattern?







A is correct. The concrete component in B also has a base Component as attribute, but this is not allowed as it can only be a Base. In C, the Concrete Decorators are extending the Concrete Component, but they should be extending the Base Decorators.

Question 10

Suppose we have a data structure representing multiple vehicles (e.g. car, bus, truck etc) with the customization option to have extra large tires. There are no further customization options.

Should we use the Decorator Pattern here and how?

A. Yes, the vehicles are the Concrete Components and the extra large tires the Decorators

B. Yes, the tires are the Concrete Components and the vehicles the Decorators

C. No, it is not necessary to apply the pattern

Question 10

Suppose we have a data structure representing multiple vehicles (e.g. car, bus, truck etc) with the customization option to have extra large tires. There are no further customization options.

Should we use the Decorator Pattern here and how?

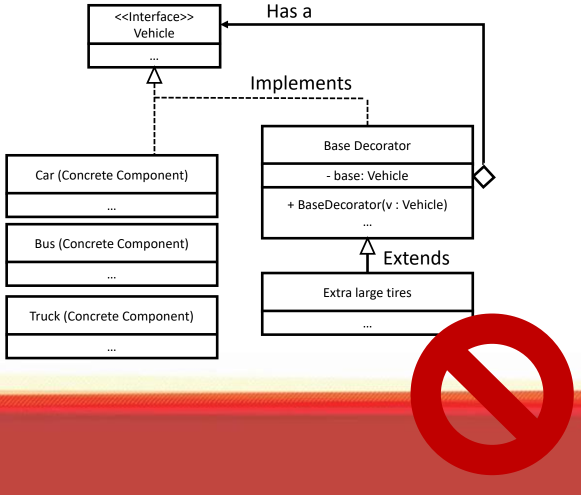
A. Yes, the vehicles are the Concrete Components and the extra large tires the Decorators

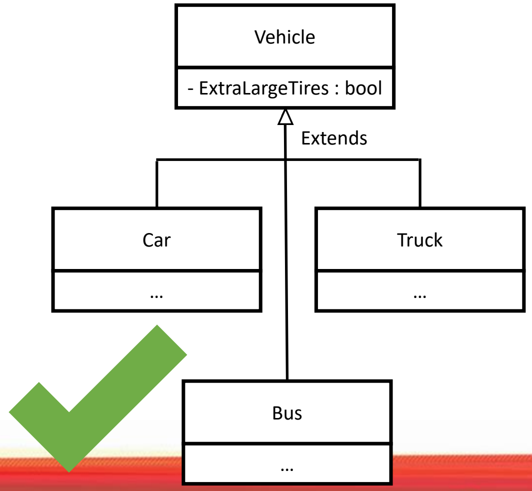
B. Yes, the tires are the Concrete Components and the vehicles the Decorators

C. No, it is not necessary to apply the pattern

We have many different Concrete Components (the vehicles) and only one Decorator (extra large tires). It is better to simply add ExtraLargeTires as a Boolean attribute to all the vehicles which is either true or false

Question 10





Question 11

Suppose we have a data structure representing a tech shop. It is possible to buy either a desktop or laptop with multiple accessories, like a mouse, mouse pad, keyboard etc. Moreover, multiple mouses or keyboards can be added to the order.

Should we use the Decorator Pattern here and how?

A. Yes, the laptop/desktop are the Concrete Components and the accessories are the Decorators

B. Yes, the accessories are the Concrete Components and the laptop/desktop are the Decorators

C. No, it is not necessary to apply the pattern

Question 11

Suppose we have a data structure representing a tech shop. It is possible to buy either a desktop or laptop with multiple accessories, like a mouse, mouse pad, keyboard etc. Moreover, multiple mouses or keyboards can be added to the order.

Should we use the Decorator Pattern here and how?

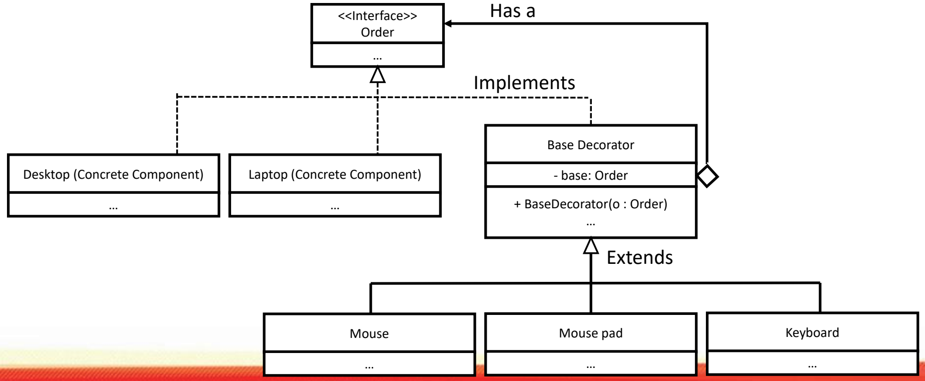
A. Yes, the laptop/desktop are the Concrete Components and the accessories are the Decorators

B. Yes, the accessories are the Concrete Components and the laptop/desktop are the Decorators

C. No, it is not necessary to apply the pattern

The Decorator pattern fits perfectly for this use case, see the next slide

Question 11



Question 12

Suppose we have a data structure representing a GUI window with the option to have a border around it. This border can be exactly one of many colors. There are no other customizable options.

Should we use the Decorator Pattern here and how?

A. Yes, the window is the Concrete Component and the border option and colors are the Decorators

B. Yes, the border option and the colors are the Concrete Components and the window is the Decorator

C. No, it is not necessary to apply the pattern

Question 12

Suppose we have a data structure representing a GUI window with the option to have a border around it. This border can be exactly one of many colors. There are no other customizable options.

Should we use the Decorator Pattern here and how?

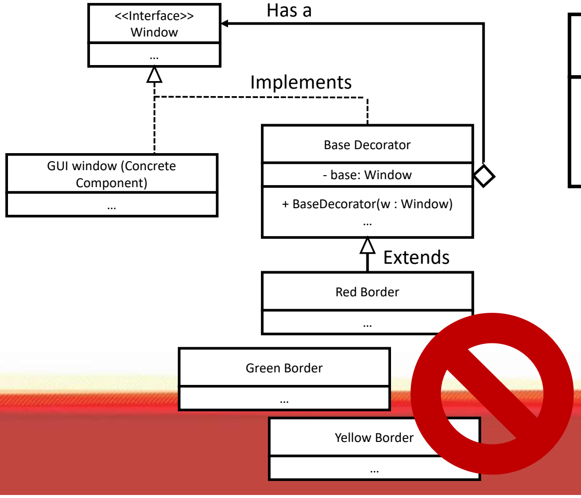
A. Yes, the window is the Concrete Component and the border option and colors are the Decorators

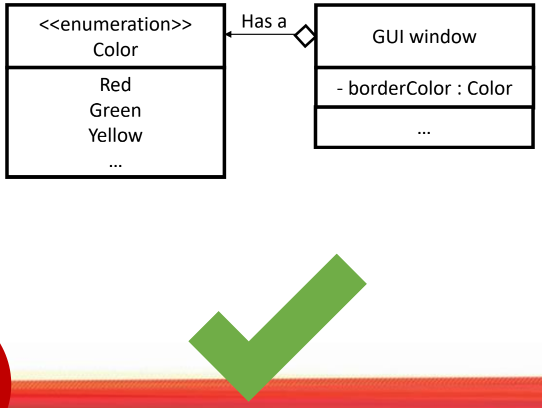
B. Yes, the border option and the colors are the Concrete Components and the window is the Decorator

C. No, it is not necessary to apply the pattern

The different colors can work as Decorators and the GUI window as a Concrete Component, but it is a bit too much as there is exactly one color used. The different colors can better be represented as an enum and the color of the border as an attribute inside the GUI window object.

Question 12



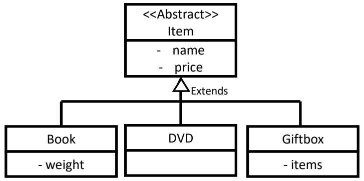


Recap

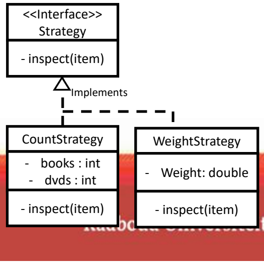
- The Decorator Pattern can be used if there are multiple properties (Decorators) that can be "wrapped" around a base object (Concrete Component)
  - E.g. options for a GUI window, accessories for products etc.
- It splits the Concrete Component and Decorators
- Multiple Decorators can be "wrapped" around a Concrete Component to create a new Component
- With this Pattern, one can easily
  - Add new Decorators
  - Add new Concrete Components

Visitor Pattern

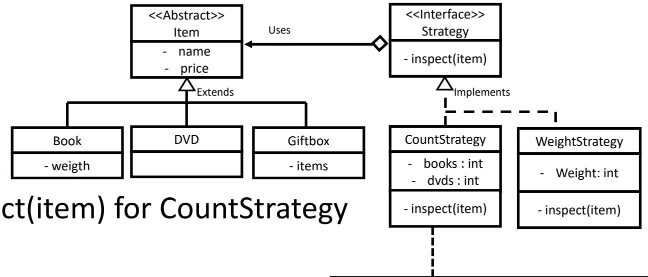
Problem I



- Suppose we have a shop with different items:
  - DVD's (name and price)
  - Books (name, price and unique weight)
  - Giftbox (name, price and list of items)
- There are different operations that can be performed on items
  - Count all the books and DVDs
  - Get the total weight of the item
- These can be seen as different "strategies"



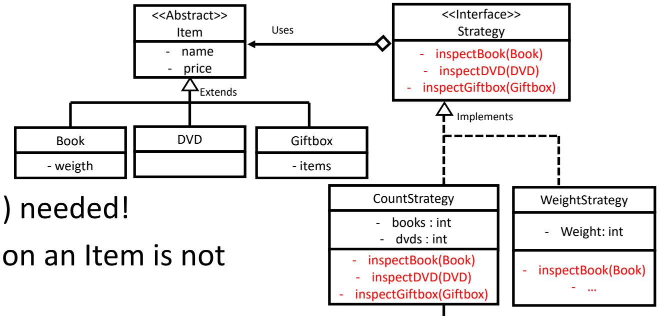
Problem II



- At first glance this look OK
- However, what does inspect(item) for CountStrategy looks like?
- Using getClass() or instanceof is not something we want in OO…
- Solution: make an inspect function for every item!

```
Void inspect(item) {
  if (item.getClass() == Book.class) {
     books += 1;
  } else if (item.getClass() == DVD.class) {
     dvds += 1;
  } else if (item.getClass() == Giftbox.class) {
     for (Item subItem: item.getItems()) {
        inspect(subItem);
     }
  }
}
```

Problem III

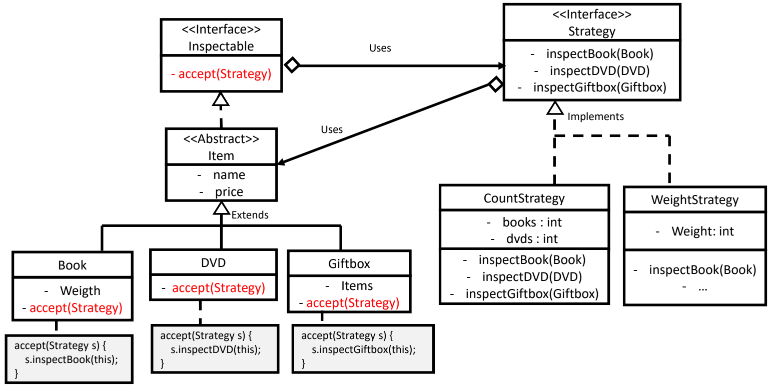


- New implementation:
- No instanceof or getClass() needed!
- Problem: using a Strategy on an Item is not abstract anymore…
- Solution: Make a new function for all items which
  - Takes a strategy as argument
  - Chooses the correct inspect function for the current Item

```
void inspectBook(Book b) {
   books += 1;
}

void inspectDVD(DVD d) {
   dvds += 1;
}

Void inspectGiftbox(Giftbox gb) {
  for (Item subItem: item.getItems()) {
      inspect(subItem);
  }
}
```

Problem IV

- Implementation:



```
accept(Strategy s) {
  s.inspectBook(this);
}
```

```
accept(Strategy s) {
  s.inspectDVD(this);
}
```

```
accept(Strategy s) {
  s.inspectGiftbox(this);
}
```

- The question probably arises: why would we want this complex pattern?
  - For every item and strategy, we can use item.accept(strategy) and it will work
  - No usage of instanceof or getClass()!
  - The Inspectable interface doesn't have to be touched anymore, but many strategies can be added!

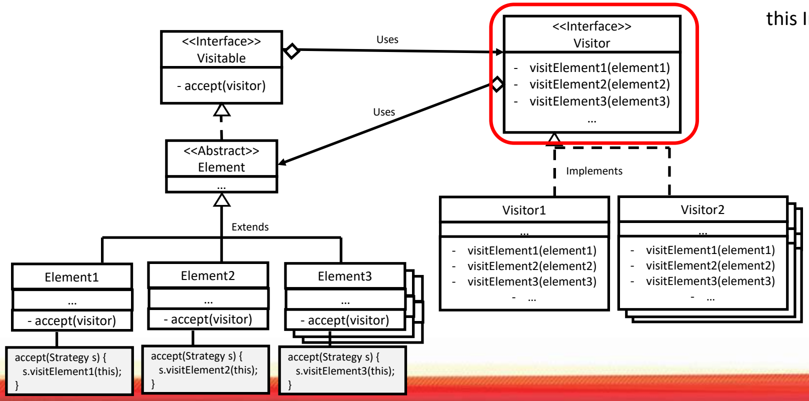
Visitor Pattern

- Visitor Pattern is very similar to the Strategy Pattern
- But also works with multiple objects!
  - While keeping the code clean
- Objects are **Visitable** and contain the accept(strategy) function
- Strategies are **Visitors** and contain a visitObject(object) function for every Object
  - These are linked with each other
- Really useful when the Visitable Objects do not change often
- Many Visitors can be added and the code for the Visitables is unchanged!

Visitor Pattern UML

<<Interface>> Visitor

- The interface for the Visitors, which contain a visit function for every element
- All the concrete Visitors implement this Interface



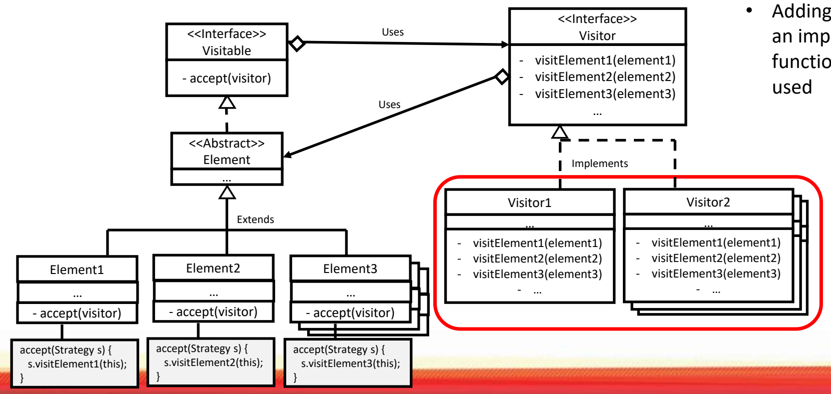
<<Interface>>
Visitable
- accept(visitor)
Uses
<<Interface>>
Visitor
- visitElement1(element1)
- visitElement2(element2)
- visitElement3(element3)
...
Uses
<<Abstract>>
Element
...
Implements
Extends
Visitor1
...
- visitElement1(element1)
- visitElement2(element2)
- visitElement3(element3)
- ...
Visitor2
...
- visitElement1(element1)
- visitElement2(element2)
- visitElement3(element3)
- ...
Element1
...
- accept(visitor)
Element2
...
- accept(visitor)
Element3
...
- accept(visitor)

```
accept(Strategy s) {
  s.visitElement1(this);
}
```

```
accept(Strategy s) {
  s.visitElement2(this);
}
```

```
accept(Strategy s) {
  s.visitElement3(this);
}
```

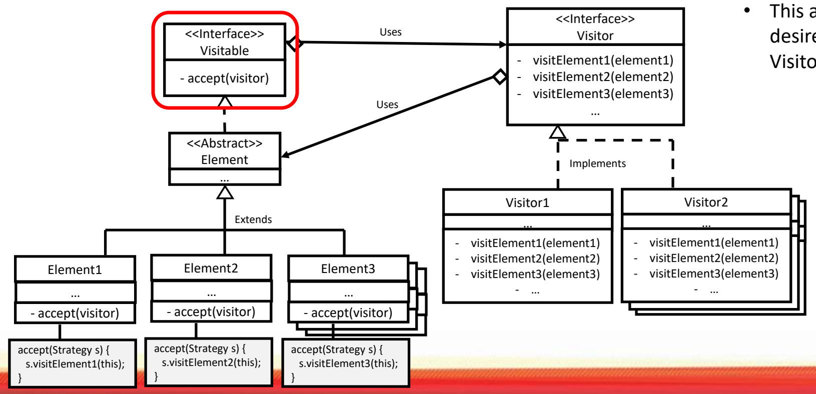
Visitor Pattern UML



Concrete Visitors

- These are the concrete Visitors with their own implementation of the visit function for all different elements
- Adding a new Visitor only requires an implementation of these visit functions and it can immediately be used

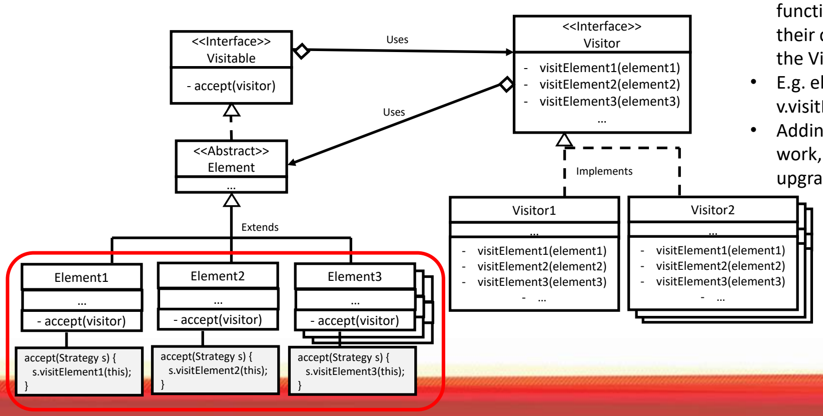
Visitor Pattern UML



<<Interface>> Visitable

- The interface for all the Visitable Elements.
- All implementations have an accept function that takes a Visitor
- This accept function executes the desired behavior from the passed Visitor on the Visitable Element

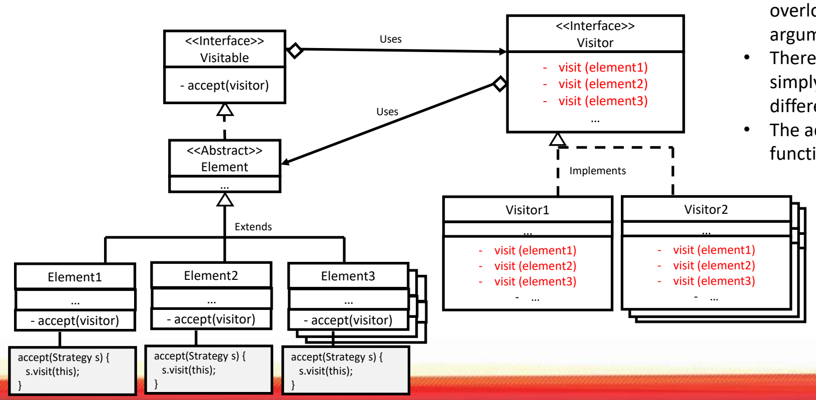
Visitor Pattern UML



Concrete Elements

- The concrete elements used in the program which extend the Abstract Element
- These all implement an accept function which is directly linked to their corresponding visit function in the Visitor
- E.g. el3.accept(visitor v) calls v.visitElement3(el3)
- Adding a new Element is a bit more work, as all Visitors should be upgraded with a new visit function

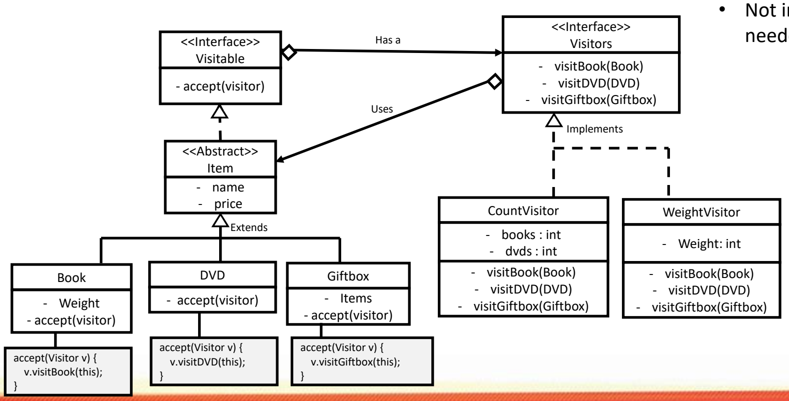
Visitor Pattern UML



Overloading

- Before, we used visitElement1(element1), visitElement2(element2) etc
- However, these functions can be overloaded, as the type of the arguments are different
- Therefore, it is common practice to simply use visit as the name of the different functions
- The accept(strategy) is the same function everywhere!

Application on previous problem



<<Interface>>
Visitable

- accept(visitor)

Has a

<<Interface>>
Visitors

- visitBook(Book)
- visitDVD(DVD)
- visitGiftbox(Giftbox)

Uses

Implements

<<Abstract>>
Item

- name
- price

Extends

Book

- Weight
- accept(visitor)

DVD

- accept(visitor)

Giftbox

- Items
- accept(visitor)

CountVisitor

- books : int
- dvds : int

- visitBook(Book)
- visitDVD(DVD)
- visitGiftbox(Giftbox)

WeightVisitor

- Weight: int

- visitBook(Book)
- visitDVD(DVD)
- visitGiftbox(Giftbox)

```
accept(Visitor v) {
  v.visitBook(this);
}
```

```
accept(Visitor v) {
  v.visitDVD(this);
}
```

```
accept(Visitor v) {
  v.visitGiftbox(this);
}
```

- As can be seen, for every item and every Visitor v, item.accept(v) can immediately be applied
- Not instanceof or getClass() is needed anymore in the Visitors!

Live coding session

- Lets take a look at the program from the previous example
- This can be found in VisitorPatternExample.zip
- Take a look at it in your own time as well and experiment!

Question 13

Suppose we have a program for an electrical business that sells keyboards, mouses, desktops etc and a complete set of these attributes. Different operations can be performed on these objects, e.g. getting the price, the operation system requirements etc.

Should we use the Visitor pattern and how?

A. Yes, the products are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the products the Visitables

C. No, it is not necessary to apply the pattern

Question 13

Suppose we have a program for an electrical business that sells keyboards, mouses, desktops etc and a complete set of these attributes. Different operations can be performed on these objects, e.g. getting the price, the operation system requirements etc.

Should we use the Visitor pattern and how?

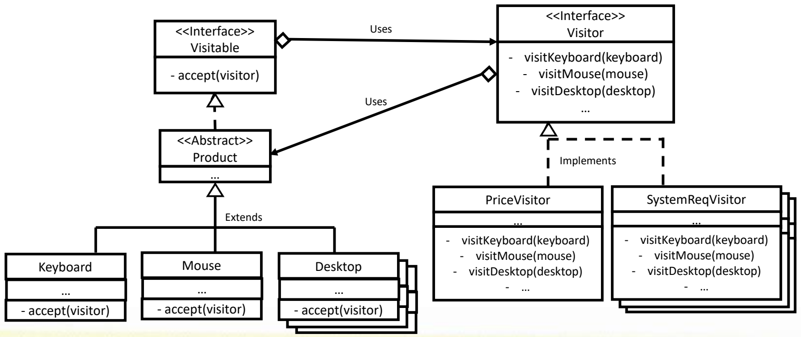
A. Yes, the products are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the products the Visitables

C. No, it is not necessary to apply the pattern

The Visitor pattern fits perfectly in this use case, see the next slide

Question 13



<<Interface>> Visitable

- accept(visitor)

Uses

<<Interface>> Visitor

- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)

...

Uses

<<Abstract>> Product

...

Implements

PriceVisitor

...

- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)
- ...

SystemReqVisitor

...

- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)
- ...

Extends

Keyboard

...

- accept(visitor)

Mouse

...

- accept(visitor)

Desktop

...

- accept(visitor)

Question 14

Suppose we have a shop that sells bikes. All these bikes are in essence the same, but they differ in their attributes, e.g. different height, different tires, different colors. Several operations should be applied on these bikes, e.g. calculating the production cost or the total weight of the bike.

Should we use the Visitor pattern and how?

A. Yes, the bikes are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the bikes the Visitables

C. No, it is not necessary to apply the pattern

Question 14

Suppose we have a shop that sells bikes. All these bikes are in essence the same, but they differ in their attributes, e.g. different height, different tires, different colors. Several operations should be applied on these bikes, e.g. calculating the production cost or the total weight of the bike.

Should we use the Visitor pattern and how?

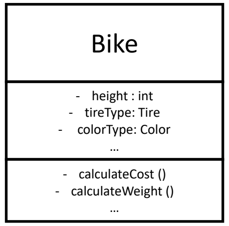
A. Yes, the bikes are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the bikes the Visitables

C. No, it is not necessary to apply the pattern

There is only one Visitable object, namely the bike which has multiple attributes. The Visitor pattern is not useful here, as the operations that should be performed on the bike should be implemented in that class. See the next slide

Question 14



Bike

- height : int
- tireType: Tire
- colorType: Color

...

- calculateCost ()
- calculateWeight ()

...

Question 15

Suppose we have a map with streets, roundabouts, bridges etc. Multiple calculations can be performed on these roads, e.g. calculating the length of a route or the fuel consumption. Other calculations could be added to the program, and different maps can be made and will extend the different road objects with roads specifically for bikes or trains for example.

Should we use the Visitor pattern and how?

A. Yes, the road objects are the Visitables and the calculations the Visitors

B. Yes, the calculations are the Visitables and the road objects the Visitors

C. No, it is not necessary to apply the pattern

Question 15

Suppose we have a map with streets, roundabouts, bridges etc. Multiple calculations can be performed on these roads, e.g. calculating the length of a route or the fuel consumption. Other calculations could be added to the program, and different maps can be made and will extend the different road objects with roads specifically for bikes or trains for example.

Should we use the Visitor pattern and how?

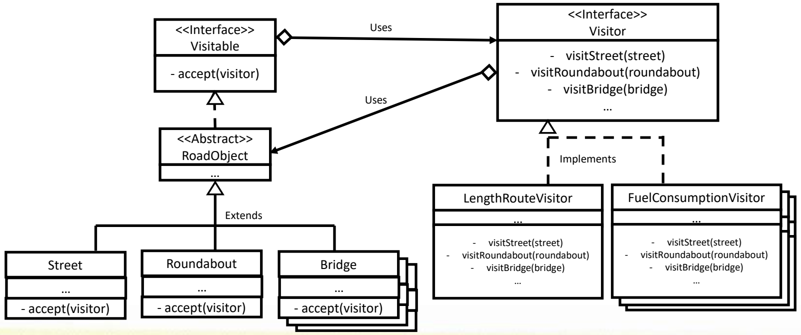
A. Yes, the road objects are the Visitables and the calculations the Visitors

B. Yes, the calculations are the Visitables and the road objects the Visitors

C. No, it is not necessary to apply the pattern

The Visitor pattern can be applied here. Note that it is a bit more work to add new road objects, but the pattern is still viable. See the next slide

Question 15



Recap

- The Visitor pattern can be used to split up several algorithms from a lot of different Elements with different properties
  - A lot of different problems fall into this category!
- It puts the Elements in a Visitable Interface and the algorithms in a Visitor interface
  - The visitors (algorithms) can visit the Visitables (Element)
- With this pattern, one can easily
  - Add new Visitors
- Note that it a bit more work to add new Elements
  - This pattern is used best when the Elements do not change a lot

Recognising Patterns

Recognising Patterns

- You have seen different patterns
  - But there are many more!
  - You probably used a lot of these subconsciously
- Learning these patterns by heart is not the goal
- Recognising reappearing structures in a program is the goal!
  - And we named these “reappearing structures”
- Using the design patterns benefits your code
  - Better readability
  - Better maintainability
- However, do not overuse (or misuse) these patterns if not necessary!

Question 16

In Brightspace, there are different roles: teachers, students, TA’s etc. Each TA can have different rights in Brightspace; some can grade stuff, some can add material, some can only view all material. These different rights can be combined with each other. The teacher can assign rights to each individual TA.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 16

In Brightspace, there are different roles: teachers, students, TA’s etc. Each TA can have different rights in Brightspace; some can grade stuff, some can add material, some can only view all material. These different rights can be combined with each other. The teacher can assign rights to each individual TA.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

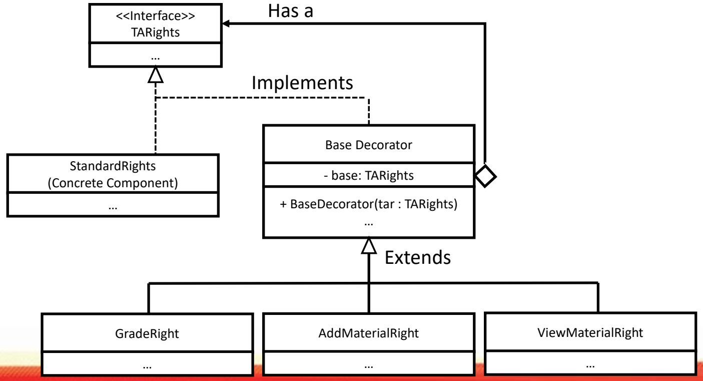
B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

The standard rights for TA’s is the Concrete Component and the other different rights which are not standard are the Decorators. It is also possible to implement the different rights as Boolean attributes. See the next slide

Question 16



Question 17

In Brightspace, we have different views for a course page, e.g. students can only see material that is published and TA’s can see all material, but cannot see the menu for adding or removing material. Teachers can see everything, including this menu. We are looking at the part of the program that shows these different views/interfaces on the screen.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 17

In Brightspace, we have different views for a course page, e.g. students can only see material that is published and TA’s can see all material, but cannot see the menu for adding or removing material. Teachers can see everything, including this menu. We are looking at the part of the program that shows these different views/interfaces on the screen.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

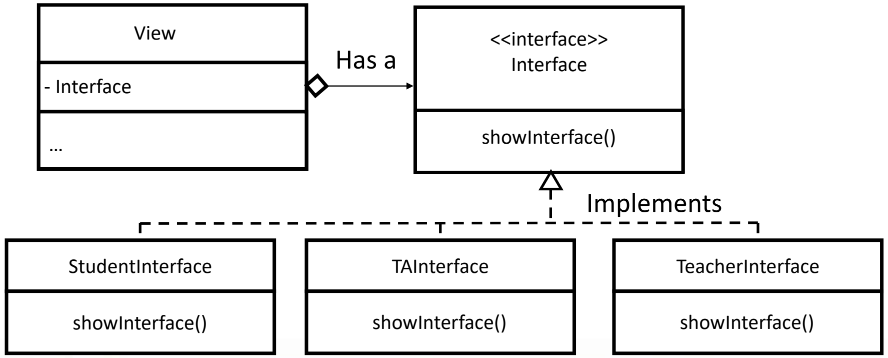
B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

These different views can be seen as different strategies, which results in an application of the strategy pattern. See the next slide.

Question 17



Question 18

In Brightspace, students should be notified when grades are published or the teacher places an announcement for all students. The other way around, the teacher should receive a message on Brightspace whenever a students asks something in a discussion thread or hands in an assignment.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 18

In Brightspace, students should be notified when grades are published or the teacher places an announcement for all students. The other way around, the teacher should receive a message on Brightspace whenever a students asks something in a discussion thread or hands in an assignment.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

No pattern that we discussed in this lecture can be applied here, but this is a typical example of the Observer Pattern, which you probably already used in different programs. This Pattern waits till something specific happens (grades are published) and then executes a certain action (notify the students).

Recap

- We discussed 3 concrete design patterns
  - Strategy pattern
  - Decorator pattern
  - Visitor pattern
- For each pattern, we saw
  - How it works
  - An implementation in Java
  - The use cases
- In your assignment, you have to
  - Recognise a pattern in given code and refactor this code
  - Implement the visitor pattern

Questionnaire

- Please fill in this questionnaire for my research (takes at most 2 minutes, but it really helps me to research/improve education)
- Also if you are watching the recording or reading the slides!

References for the specific patterns

Strategy Pattern:

https://refactoring.guru/design-patterns/strategy

Decorator Pattern:

https://refactoring.guru/design-patterns/decorator

Visitor Pattern:

https://refactoring.guru/design-patterns/visitor

I used these sites as my main reference for the examples. They are very clear and easy to follow.

References for design patterns

Clear and simple explanations of patterns:

https://refactoring.guru/design-patterns/catalog

Short explanations of the patterns with one simple, elaborated example:

https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm

A bit longer and in-depth explanations of patterns:

https://www.oodesign.com/

Several design patterns discussed with a real-life example:

https://ronnieschaniel.medium.com/object-oriented-design-patterns-explained-using-practical-examples-84807445b092

Appendix B

Online quiz

socrative

Name ____________
Date ____________

Quiz Design Patterns

Score ____________

1. Which code fragment is represented by this UML?

[✓ Ⓐ] Code fragment A

Ⓑ Code fragment B

Ⓒ Code fragment C

Ⓓ Code fragment D

2. Which UML represents this code fragment?

Ⓐ UML A

Ⓑ UML B

[✓ Ⓒ] UML C

Ⓓ UML D

3. Which of the following programs have a similar structure?
1. Getting the last names of a list of students
2. Evaluating the value of a given boolean formula
3. For the numbers between 1 and 100, calculate the square root

Ⓐ 1 and 2 have a similar structure

[✓ Ⓑ] 1 and 3 have a similar structure

Ⓒ 2 and 3 have a similar structure

Ⓓ None of these programs have a similar structure

4. Which of the following objects have a similar structure?
1. Many different bikes with an option to add a specific, special saddle
2. A pizza with many options for different toppings, e.g. tomato, cheese, chicken etc
3. A server where users can have a combination of different rights, e.g. reading, writing, editing, adding new users etc

Ⓐ 1 and 2 have a similar structure

Ⓑ 1 and 3 have a similar structure

[✓ Ⓒ] 2 and 3 have a similar structure

Ⓓ None of these programs have a similar structure

5. Which is the correct UML for the Strategy Pattern?

[✓ Ⓐ] UML A

Ⓑ UML B

Ⓒ UML C

Ⓓ UML D

6. Suppose the following program:
We have a program that encrypts a given message with either RSA or
Triple DES (2 encryption methods). The program probably won’t be
extended with other encryption methods.
Should we use the strategy pattern and how?

[✓ Ⓐ] Yes, the encryption methods are the strategies

Ⓑ Yes, the different messages are the strategies

[✓ Ⓒ] No, it is not necessary to apply the pattern

7. Suppose the following program:
We have a simulation of robots which move in a field. These robots
have different behaviors, e.g. defensive, offensive or random. These
behaviors can also be combined with each other or can have different variations.
Should we use the strategy pattern and how?

Ⓐ Yes, the robots are the strategies

[✓ Ⓑ] Yes, the behaviors are the strategies

Ⓒ No, it is not necessary to apply the pattern

8. Suppose the following program:
We have a program that extracts plain text from different types of files, e.g. pdf, xml or html.
This is then returned as a string.
Should we use the strategy pattern and how?

Ⓐ Yes, the different types of files are the strategies

Ⓑ Yes, the different extraction algorithms are the strategies

[✓ Ⓒ] No, it is not necessary to apply the pattern

9. Which is the correct UML for the Decorator Pattern?

[✓ Ⓐ] UML A

Ⓑ UML B

Ⓒ UML C

Ⓓ UML D

10. Suppose we have a data structure representing multiple vehicles (e.g. car, bus, truck etc) with the customization option to have extra large tires. There are no further customization options.
Should we use the Decorator Pattern here and how?

Ⓐ Yes, the vehicles are the Concrete Components and the extra large tires the Decorators

Ⓑ Yes, the tires are the Concrete Components and the vehicles the Decorators

[✓ Ⓒ] No, it is not necessary to apply the pattern

11. Suppose we have a data structure representing a tech shop. It is possible to buy either a desktop or laptop with multiple accessories, like a mouse, mouse pad, keyboard etc.
Should we use the Decorator Pattern here and how?

[✓ Ⓐ] Yes, the laptop/desktop are the Concrete Components and the accessories the Decorators

Ⓑ Yes, the accessories are the Concrete Components and the laptop/desktop are the Decorators

Ⓒ No, it is not necessary to apply the pattern

12. Suppose we have a data structure representing a GUI window with the option to have a border around it. This border can be exactly one of many colors. There are no other customizable options.
Should we use the Decorator Pattern here and how?

Ⓐ Yes, the window are the Concrete Component and the border option and colors are the Decorators

Ⓑ Yes, the border option and the colors are the Concrete Components and the window is the Decorator

[✓ Ⓒ] No, it is not necessary to apply the pattern

13. Suppose we have a program for an electrical business that sells keyboards, mouses, desktops etc and a complete set of these attributes. Different operations can be performed on these objects, e.g. getting the price, the operation system requirements etc.
Should we use the Visitor pattern and how?

[✓ Ⓐ] Yes, the products are the Visitables and the operations the Visitors

Ⓑ Yes, the operations are the Visitables and the products the Visitables

Ⓒ No, it is not necessary to apply the pattern

14. Suppose we have a shop that sells bikes. All these bikes are in essence the same, but they differ in their attributes, e.g. different height, different tires, different colors. Several operations should be applied on these bikes, e.g. calculating the production cost or the total weight of the bike.
Should we use the Visitor pattern and how?

Ⓐ Yes, the bikes are the Visitables and the operations the Visitors

Ⓑ Yes, the operations are the Visitables and the bikes the Visitables

[✓ Ⓒ] No, it is not necessary to apply the pattern

15. Suppose we have a map with streets, roundabouts, bridges etc. Multiple calculations can be performed on these roads, e.g. calculating the length of a route or the fuel consumption. Other
calculations could be added to the program, and different maps can be made and will extend the different road objects with roads specifically for bikes or trains for example.
Should we use the Visitor pattern and how?

[✓ Ⓐ] Yes, the road objects are the Visitables and the calculations the Visitors

Ⓑ Yes, the calculations are the Visitables and the road objects the Visitables

Ⓒ No, it is not necessary to apply the pattern

16. In Brightspace, there are different roles: teachers, students, TA's etc. Each TA can have different rights in Brightspace; some can grade stuff, some can add material, some can only view all
material. These different rights can be combined with each other. The teacher can assign rights to each individual TA.
Which design pattern can be recognised and applied here?

Ⓐ Strategy Pattern

[✓ Ⓑ] Decorator Pattern

Ⓒ Visitor Pattern

Ⓓ No pattern or undiscussed pattern

17. In Brightspace, we have different views for a course page, e.g. students can only see material that is published and TA's can see all material, but cannot see the menu for adding or removing
material. Teachers can see everything, including this menu. We are looking at the part of the program that shows these different views/interfaces on the screen.
Which design pattern can be recognised and applied here?

[✓ Ⓐ] Strategy Pattern

Ⓑ Decorator Pattern

Ⓒ Visitor Pattern

Ⓓ No pattern or undiscussed pattern

18. In Brightspace, students should be notified when grades are published or the teacher places an announcement for all students. The other way around, the teacher should receive a message on
Brightspace whenever a students asks something in a discussion thread or hands in an assignment.
Which design pattern can be recognised and applied here?

Ⓐ Strategy Pattern

Ⓑ Decorator Pattern

Ⓒ Visitor Pattern

[✓ Ⓓ] No pattern or undiscussed pattern

Appendix C

Live coding sessions

C.1 Strategy Pattern

```
public class StrategyPatternExample {

    public static void main(String[] args) {
        Context context = new Context();
        // The normal routing algorithm is used
        context.setRa(new NormalRoute());
        System.out.println(context.returnRoute("A","D"));
        // Only car routes can be used
        context.setRa(new CarRoute());
        System.out.println(context.returnRoute("A","D"));
        System.out.println(context.returnRoute("D","B"));
        // All routes can be used, but the route should go via D
        context.setRa(new RouteViaD());
        System.out.println(context.returnRoute("A","C"));
        System.out.println(context.returnRoute("A","A"));
    }

}
```

```
import java.util.Arrays;
import java.util.LinkedList;

public class Context {
    // The used Strategy
    private RoutingAlgorithm ra;
    // The used map for the strategy
    public final Map roadmap = new Map(new
        LinkedList<>(Arrays.asList(
        new Edge(4, new String[] {"Car","Walk"}, "A", "B"),
        new Edge(8, new String[] {"Car","Walk"}, "B", "C"),
        new Edge(7, new String[] {"Car","Walk"}, "C", "D"),
        new Edge(10, new String[] {"Car","Walk"}, "D", "A"),
        new Edge(3, new String[] {"Walk"}, "A", "C"),
        new Edge(2, new String[] {"Walk"}, "B", "D"))));

    // Returns the shortest route on the attribute roadmap with the
        given start and end point
    public Route returnRoute(String startpoint, String endpoint) {
        return ra.calcRoute(roadmap, startpoint, endpoint);
    }

    public void setRa(RoutingAlgorithm ra) {
        this.ra = ra;
    }


}
```

```
// The strategies for the routing algorithms

// Each contains a function to calculate a route with a given map,
    starting point and end point
public interface RoutingAlgorithm {
    public Route calcRoute(Map map, String start, String end);
}

// This strategy takes all roads into account
class NormalRoute implements RoutingAlgorithm {
    @Override
    public Route calcRoute(Map map, String start, String end) {
        return map.calcShortestFromTo(start, end);
    }
}

// This strategy only takes car roads into account
class CarRoute implements RoutingAlgorithm {
    @Override
    public Route calcRoute(Map map, String start, String end) {
        Map newMap = map.onlyRoadsWith("Car");
        return newMap.calcShortestFromTo(start, end);
    }
}

class RouteViaD implements RoutingAlgorithm {
    @Override
    public Route calcRoute(Map map, String start, String end) {
        Route route1 = map.calcShortestFromTo(start, "D");
        Route route2 = map.calcShortestFromTo("D", end);
        route1.addRoute(route2);
        return route1;
    }
}
```

```
import java.util.Arrays;
import java.util.LinkedList;

// The Class that represents a map with some routing algorithms
// Not relevant for the students
public class Map {
    private LinkedList<Edge> roads;

    public Map(LinkedList<Edge> roads) {
        this.roads = roads;
    }

    private Route calcShortestFromToRec(String start, String end,
        int iteration) {
        if (iteration >= 4)
            return null;
        if (start.equals(end)) {
            LinkedList<String> list = new LinkedList<>();
            list.add(start);
            return (new Route(list, 0));
        }
        iteration+=1;
        LinkedList<Route> routes = new LinkedList<>();
        if (findEdgeFromTo(start, "A") != null &&
            !"A".equals(start)) {
            Route imroute = calcShortestFromToRec("A", end,
                iteration);
            if (imroute != null) {
                imroute.updateWithNodeAndLength(start,
                    findEdgeFromTo(start, "A").getLength());
                routes.add(imroute);
            }
        }
        if (findEdgeFromTo(start, "B") != null &&
            !"B".equals(start)) {
            Route imroute = calcShortestFromToRec("B", end,
                iteration);
            if (imroute != null) {
                imroute.updateWithNodeAndLength(start,
                    findEdgeFromTo(start, "B").getLength());
                routes.add(imroute);
            }
        }
        if (findEdgeFromTo(start, "C") != null &&
            !"C".equals(start)) {
            Route imroute = calcShortestFromToRec("C", end,
                iteration);
            if (imroute != null) {
                imroute.updateWithNodeAndLength(start,
                    findEdgeFromTo(start, "C").getLength());
                routes.add(imroute);
            }
        }
        if (findEdgeFromTo(start, "D") != null &&
            !"D".equals(start)) {
            Route imroute = calcShortestFromToRec("D", end,
                iteration);
            if (imroute != null) {
                imroute.updateWithNodeAndLength(start,
                    findEdgeFromTo(start, "D").getLength());
                routes.add(imroute);
            }
        }
        return getBestRoute(routes);
    }

    private Route getBestRoute(LinkedList<Route> routes) {
        if (routes.isEmpty())
            return null;
        Route bestRoute = routes.get(0);
        for (int i = 1; i < routes.size(); i++) {
            if(routes.get(i).getLength() < bestRoute.getLength())
                bestRoute = routes.get(i);
        }
        return bestRoute;
    }

    public Route calcShortestFromTo(String start, String end) {
        return calcShortestFromToRec(start, end, 0);
    }

    private Edge findEdgeFromTo(String start, String end) {
        for(Edge e: roads) {
            String points = e.getPoints();
            if (points.equals(start + " " + end) ||
                points.equals(end + " " + start))
                return e;
        }
        return null;
    }

    public Map onlyRoadsWith(String vehicle) {
        LinkedList<Edge> result = new LinkedList<>();
```

```
        for(Edge e : roads) {
            if(Arrays.asList(e.getVehicles()).contains(vehicle))
                result.add(e);
        }
        return new Map(result);
    }

    @Override
    public String toString() {
        return "Map{" + "roads=" + roads + ’}’;
    }


}
```

```
import java.util.Arrays;

// The Class that represents the Edges on the Map
// Not relevant for the students
public class Edge {
    private int length;
    private String[] vehicles;
    private String point1;
    private String point2;

    public Edge(int length, String[] vehicles, String point1,
        String point2) {
        this.length = length;
        this.vehicles = vehicles;
        this.point1 = point1;
        this.point2 = point2;
    }

    public String getPoints(){
        return (point1 + " " + point2);
    }

    public int getLength() {
        return length;
    }

    public String[] getVehicles() {
        return vehicles;
    }

    @Override
    public String toString() {
        return "Edge{" + "length=" + length + ", vehicles=" +
            Arrays.toString(vehicles) + ", point1=" + point1 + ",
            point2=" + point2 + ’}’;
    }


}
```

```
import java.util.LinkedList;

// The Class that represents routes
// Not relevant for the students
public class Route {
    private LinkedList<String> nodes;
    private int length;

    public Route (LinkedList <String> nodes, int length) {
        this.nodes = nodes;
        this.length = length;
    }

    public void updateWithNodeAndLength(String node, int length) {
        nodes.offerFirst(node);
        this.length += length;
    }

    public void addRoute(Route route){
        LinkedList<String> additroutes = route.getNodes();
        additroutes.remove();
        this.nodes.addAll(additroutes);
        this.length += route.getLength();
    }

    public int getLength() {
        return length;
    }

    public LinkedList<String> getNodes() {
        return nodes;
    }



    @Override
    public String toString() {
        return "Route{" + "nodes=" + nodes + ", length=" + length +
            ’}’;
    }
}
```

C.2 Decorator Pattern

```
public class DecoratorPatternExample {
    public static void main(String[] args) {
        System.out.println("Notifier 1:");
        // Only the basic Notifier
        Notifier n1 = new EmailNotifier();
        n1.notifyUser();
        System.out.println("Notifier 2:");
        // A basic Notifier with Whatsapp and Instagram as well
        Notifier n2 = new WhatsappNotifier(new InstagramNotifier(new
            EmailNotifier()));
        n2.notifyUser();
    }

}
```

```
// The Notifier interface
public interface Notifier {
    public void notifyUser();
}
```

```
// The Concrete Email Notifier
// Note that is does not contain another Notifier inside it, is
    this is the wrappee and not a wrapper
public class EmailNotifier implements Notifier {
    // This notifyUser gives an email notifier
    @Override
    public void notifyUser() {
        System.out.println("Notifying via mail");
        // Notify via mail
    }
}
```

```
// The Base Decorator for the program
public class BaseNotifierDecorator implements Notifier {
    private Notifier base;

    // Constructor simply sets the attribute base Notifier
    public BaseNotifierDecorator(Notifier n) {
        this.base = n;
    }
    // NotifyUser simply calls the notifyUser function on the
        stored base
    @Override
    public void notifyUser() {
        base.notifyUser();
    }
}
```

```
// The Decorator for the Whatsapp Notifier
public class WhatsappNotifier extends BaseNotifierDecorator {
    // Super simply calls the Constructor of the parent class
    // In this case BaseNotifierDecorator, so it simply sets the
        base
    public WhatsappNotifier (Notifier n) {
        super(n);
    }

    // This notifyUser notifies a user via Whatsapp
    @Override
    public void notifyUser() {
        super.notifyUser();
        System.out.println("Notifying via Whatsapp");
        // Notify via Whatsapp
    }
}
```

```
// The Decorator for the SMS Notifier
public class SMSNotifier extends BaseNotifierDecorator {
    // Super simply calls the Constructor of the parent class
    // In this case BaseNotifierDecorator, so it simply sets the
        base
    public SMSNotifier (Notifier n) {
        super(n);
    }

    // This notifyUser notifies a user via SMS
    @Override
    public void notifyUser() {
        super.notifyUser();
        System.out.println("Notifying via SMS");
        // Notify via SMS
    }
}
```

```
// The Decorator for the Instagram Notifier
public class InstagramNotifier extends BaseNotifierDecorator {
    // Super simply calls the Constructor of the parent class
    // In this case BaseNotifierDecorator, so it simply sets the
        base
    public InstagramNotifier (Notifier n) {
        super(n);
    }

    // This notifyUser notifies a user via Instagram
    @Override
    public void notifyUser() {
        super.notifyUser();
        System.out.println("Notifying via Instagram");
        // Notify via Instagram
    }
}
```

C.3 Visitor Pattern

```
import java.util.Arrays;
import java.util.List;

public class VisitorPatternExample {

    public static void main(String[] args) {
        // Creating the objects
        Book book1 = new Book("Alice in Wonderland", 15, 3);
        Book book2 = new Book("Bob in Wonderland", 10, 2);
        Book book3 = new Book("Eve in Wonderland", 18, 5);
        DVD dvd1 = new DVD("The Beatles", 5);
        DVD dvd2 = new DVD("ABBA", 7);
        List<Item> items = Arrays.asList(book1, book2, book3, dvd1,
            dvd2);
        Giftbox giftbox1 = new Giftbox("giftbox", items);

        // A simple example to illustrate the inner working of the
            accept function
        Visitor v1 = new CountVisitor();
        book1.accept(v1);
        System.out.println(v1);

        // As there are 3 books and 2 DVDs in the giftbox, that will
            be the result
        Visitor v2 = new CountVisitor();
        giftbox1.accept(v2);
        System.out.println(v2);

        // As the books have a total weight of 10,
        // the DVDs both have a standard weight of 1.
        // and the box has an additional weight of 2,
        // the total weight will be 14
        Visitor v3 = new WeightVisitor();
        giftbox1.accept(v3);
        System.out.println(v3);
    }

}
```

```
public interface Visitable {
    // Only contians the accept function
    public void accept(Visitor v);
}
```

```
public abstract class Item implements Visitable {
    // All items have a name and a price
    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public int getPrice() {
        return price;
    }

}
```

```
public class Book extends Item {
     // Books do have an additional weight as an attribute
     private int weight;

     public Book(String name, int price, int weight) {
         super(name, price);
         this.weight = weight;
     }

     // The accept function simply refers to the corresponding visit
         function in the Visitor
     @Override
     public void accept(Visitor v){
         v.visitBook(this);
     }
```

```
    public int getWeight() {
        return weight;
    }

}
```

```
public class DVD extends Item {

    public DVD (String name, int price) {
        super(name,price);
    }

    // The accept function simply refers to the corresponding visit
        function in the Visitor
    @Override
    public void accept(Visitor v) {
        v.visitDVD(this);
    }

}
```

```
import java.util.List;

public class Giftbox extends Item {
    // Giftboxes contain a list of items
    private List<Item> items;

    public Giftbox (String name, List<Item> items) {
        // super calls the Object above the current object, Item in
            this case
        // So the Item constructor is called with the given name and
            the calculated price
        super(name, getTotalPrice(items));
        this.items = items;
    }

    // Function to calculate the total weight of a list of items
    private static int getTotalPrice(List<Item> items) {
        int result = 0;
        for(Item i : items) {
            result += i.getPrice();
        }
        return result;
    }

    // The accept function simply refers to the corresponding visit
        function in the Visitor
    @Override
    public void accept(Visitor v) {
        v.visitGiftbox(this);
    }

    public List<Item> getItems() {
        return items;
    }

}
```

```
public interface Visitor {
    // Contain a specific visit function for each object
    public void visitBook(Book b);
    public void visitDVD(DVD d);
    public void visitGiftbox(Giftbox g);

}
```

```
public class CountVisitor implements Visitor {
    // This Visitor counts the number of books and dvds
    private int books = 0;
    private int dvds = 0;

    // When a book is inspected, the number of books should increase
    @Override
    public void visitBook(Book b) {
        books += 1;
    }

    // When a DVD is inspected, the number of DVDs should increase
    @Override
    public void visitDVD(DVD d) {
        dvds += 1;
    }

    // When a giftbox is inspected, this visitor is called on all
        items with the accept functions
    @Override
    public void visitGiftbox(Giftbox g) {
        for(Item i : g.getItems()) {
            i.accept(this);
        }
    }

    @Override
    public String toString() {
        return "CountVisitor{" + "books=" + books + ", dvds=" + dvds
            + '}';
    }


}
```

```
public class WeightVisitor implements Visitor {
    // This Visitor calculates the total weight of items
    int weight = 0;

    // When a book inspected, its weight attribute is added
    @Override
    public void visitBook(Book b) {
        weight += b.getWeight();
    }

    // When a DVD is inspected, its standard weight of 1 is added
    @Override
    public void visitDVD(DVD d) {
        weight += 1;
    }

    // When a giftbox is inspected, an additional weight of 2 is
        added to the weights of all items
    // These are inspected with the accept function.
    @Override
    public void visitGiftbox(Giftbox g) {
        weight += 2; // Weight of the box
        for(Item i : g.getItems()) {
            i.accept(this);
        }
    }

    @Override
    public String toString() {
        return "WeightVisitor{" + "weight=" + weight + '}';
    }


}
```

Appendix D

Assignment PDF

Assignment Logical Formulas

Object Orientation

May 2022

This assignment consists of 3 parts: in the first part you are tasked to refactor a small program by applying a discussed design pattern. In the second part, you are asked to implement the discussed Visitor Pattern yourself. In the last part, you have to fill in a **mandatory** questionnaire after you finished the other 2 parts. It is fine if only one of your teammates fills this in.

Refactoring a program

You can find a small program in **RefactorExercise.zip**. This program has an `encrypt` function within `Encryptor` which takes a string to be encrypted with the given (simple) encryption method. The different encryption methods are represented by an enum and its functionality is implemented inside the `Encryptor` class. The different encryption methods are:

- Caesar3: this shifts all letters 3 places in the alphabet, e.g. "Abcz!" becomes "Defc!"
- ROT13: this rotates all letters 13 places in the alphabet, e.g. "Abcz!" becomes "Nopm!". Note that applying ROT13 twice, you receive the original string.
- Mirror: this mirrors the whole string, e.g. "Abcz!" becomes "!zcbA".

This may look like a good implementation at first glance, but it is not easily maintainable if many more encryption methods are added to the system. Imagine that several developers are working on different encryption methods, all in the same class. This will lead to several problems and bug fixing is even more challenging.

Therefore, we ask you to refactor this program with the help of a discussed design pattern to satisfy the following properties:

- The used encryption method is not passed on anymore via the `encrypt` function, but is an attribute in the `Encryptor` class
- Developers can work independently on different encryption methods, e.g. the different encryption methods should be in different files

- The main `Encryptor` class is left unchanged when new encryption methods are added

Your refactored program should pass all test cases, as you are not changing the behavior of the program. Please submit your refactored program with the help of the **project export**.

The Visitor Pattern

1 Learning Goals

In the second part of the assignment, we ask you to design and implement a representation for logical formulas in Java. After having completed this assignment, you should be able to:

- Implement classes to represent logical formulas where
  - A suitable interface or abstract class is defined that serves as a base for all nodes in your syntax tree.
  - Each logical connective is implemented as an extension of this base class.
- Introduce new operations using the visitor pattern.
- Utilize UML class and sequence diagrams to design an object oriented program.

2 Logical Formulas

This assignment is about representing and manipulating formulas of propositional logic.

A *logical formula* consists of the constants *true* and *false*, atomic propositions, logical operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\neg$ (not). Atomic propositions are like variables in programming languages. Atomic propositions can be any string, but we usually use single capital letters like A, B, P, Q. The abstract syntax of logical formulas is given by the following grammar.

$$\langle F \rangle ::= \textbf{true} \mid \textbf{false} \mid \texttt{Atomic} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

Examples for logical formulas are $A \Rightarrow B$, $\neg A \vee B$, $A \Rightarrow (A \vee B) \wedge (C \Rightarrow \neg D)$. In these examples A, B, C, D atomic propositions.

Formulas are represented in Java by trees, where the nodes correspond to the syntactic elements. Such trees are called *abstract syntax trees.* The leaves in such a tree correspond to constants or atomic propositions and internal nodes represent the operators. This means that some internal nodes have 2 children and others 1 child. The representation of formulas using syntax trees is similar to the numerical expressions in assignment 5. There is one interface for all the different nodes, and concrete classes that implement this interface for each type of node. There are two major differences compared to assignment 5.

1. In the numerical expression, operations were added to the base interface as abstract methods, and implemented in each subclass. In this assignment, the base interface is left unchanged. All operations are implemented as visitors.

2. A single concrete class is used to represent all binary operators. It uses the *strategy pattern to realize their different behaviors*

The visitor pattern has two ingredients. First the visitor interface, in this assignment called `FormulaVisitor`, and second an abstract method `accept` in the formula base class.

A simple base interface for logical formulas would look something like this:

```
public interface Formula {
    void accept( FormulaVisitor visitor ) ;
}
```

However, in this implementation, the Visitors **will** return something, but the type of this returned object differs per Visitor. To solve this, we will make the `FormulaVisitor` generic with a generic `Result` type and thus, the `accept` function also becomes a generic function with a `Result` type. Moreover, all the Formulas have a precedence in this assignment, so the interface also has a `getPrecendence` function (more on this in the next section). The actual interface in this assignment looks like this:

```
public interface Formula {
    public int getPrecedence();
    public <Result> Result accept( FormulaVisitor<Result> visitor ) ;
}
```

The `FormulaVisitor` interface has one `visit` method for each concrete node. The `Result` type is generic in the Formulavisitor, so all the `visit` functions will have this `Result` type. Note: The visit methods are overloaded, that's why they can all have the same name. It is possible to give them different names, like `visitNot`, `visitBinOp`, and so on.

```
public interface FormulaVisitor <Result> {
    Result visit( Not form );
    Result visit( BinaryOperator form );
    // and so on
}
```

All of the concrete nodes in the syntax tree are classes that implement `Formula`. Here is an example for a class of the *not* operator.

```java
public interface Not implements Formula {
    private Formula operand;

    public Not( Formula oper ) {
        this.operand = oper;
    }

    public Formula getOperand() {
        return operand;
    }

    public int getPrecedence() {
        return 3;
    }

    public <Result> Result accept( FormulaVisitor<Result> v) {
        return v.visit( this );
    }
}
```

Most of this definition should be self explanatory. The actual precedence system can be implemented the way you prefer. The `accept` method is boilerplate code that always looks like this.

All binary operators should be represented by a single class `BinaryOperator` which has an operator `BinOp` as an attribute. This is an example of the strategy pattern. `BinOp` should be an enum that implements `BinaryOperator<Boolean>` from the Java standard library. Every BinOp should have a string representation, a precedence, and an evaluation function.

3 Operations on Logical Formulas

You should implement two visitors for logical formulas: a pretty-printer and an evaluator.

The pretty-printer (`ShowVisitor`) should print a given formula to standard output, which is immediately returned as a `String` (thus, the generic `Result` type is `String` for this Visitor). A requirement is that this Visitor omits parentheses when it is safe. For this, every operator should have a precedence. The precedence of the operators are, from high to low: $\neg$, $\wedge$, $\vee$, $\Rightarrow$. This means that $\neg$ binds stronger than $\wedge$, and so on. To determine if parentheses are needed, you have to compare the precedence of an operator with that of its child to see if the parentheses are needed. The details are left for you to figure out. Hint: You can access both the precedence of the current operator and its children to compare both numbers. In theory, the non-operator formulas do not have a precedence, but for the assignment it is best to make sure it binds stronger than $\neg$, such that parentheses are always omitted. For operators with the same precedence, you should always print parenthesis to be safe.

The evaluator (EvaluateVisitor) evaluates the formula and returns the resulting boolean (thus the Result type is Boolean). It needs an environment for atomic propositions, just as the evaluator for numerical expression needs for variables. Where variables were assigned integers, atomic propositions are assigned Booleans. This means that the evaluator needs a Map<String,Boolean> to evaluate formulas. For example, in the environment $[P \mapsto \textbf{false}]$, the formula $\neg P$ should evaluate to **true**. As the return type of the visit functions are void, the result can be stored as an attribute in this EvaluateVisitor.

Here are some lines of code to give you inspiration. You should figure out the rest for yourself.

```
public interface FormulaVisitor<Result> {
    Result visit( Not form );
    Result visit( Atom form );
    // And so on...

    /*
    Alternatively, you could use the following names:
    Result visitNot( Not form );
    Result visitAtom( Atom form );
    And so on...
    */
}

public interface Formula {
    public int getPrecedence();
    public <Result> Result accept( FormulaVisitor<Result> v );
}

public class ShowVisitor implements FormulaVisitor<String> {
    public String visit(Atom form) {
        // ...
    }

    // And so on...
}

public class EvaluateVisitor implements FormulaVisitor<Boolean> {
    public Boolean visit(Atom form) {
        // ...
    }

    // And so on...
}
```

4 Your Tasks

In this assignment you have to draw two UML diagrams in steps 1 and 7. You have to hand in these diagrams. Either draw them on paper and submit a **photo as jpeg**, or use a UML tool and **export as pdf**. No other format will be accepted!

If you already have a UML drawing program you like, feel free to use it. Just make sure it can **export to pdf**. If you do not know which tool to use, we suggest Visual Paradigm. You can use it for free if you sign up with your email address. `https://online.visual-paradigm.com/subscribe.jsp`

1. Before you begin with your implementation, draw one UML class diagram of all the classes you need.

2. Implement the nodes for the syntax tree as described above. You need an interface and for concrete classes for the different nodes in the syntax tree. The classes should be called `Atom, Constant, BinaryOperator`, and `Not`. Hint: there can only be exactly two instances of Constant.

3. For every concrete node, add a corresponding `visit` method to `FormulaVisitor`.

4. Create a class `PrintVisitor` that implements `FormulaVisitor` and transforms a formula to a String.

5. Create a class `EvaluateVisitor` that implements `FormulaVisitor` and evaluates formulas in a given environment.

6. Implement `FormulaFactory` such that formulas can be easily generated.

7. The project template comes with test cases. Make sure that all test cases pass.

8. Draw a sequence diagram for printing the formula $\neg P$, given in the following code. Only include calls to `accept` and `visit`. Use the diagram in fig. 1 as a starting point.

```
Atom f2 = new Atom("P");
Not f1 = new Not(f2);
PrintVisitor v = new PrintVisitor();
f1.accept(v);
```

5 Mandatory Questionnaire

Please fill in this questionnaire after you are finished with the assignment: link to questionnaire

6 Submit Your Project

To submit your project, follow these steps. You have to submit three files: The java project, the class diagram, and the sequence diagram.

1. Submit the diagrams either as **jpeg or pdf**. No other format is accepted.
2. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.
3. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.

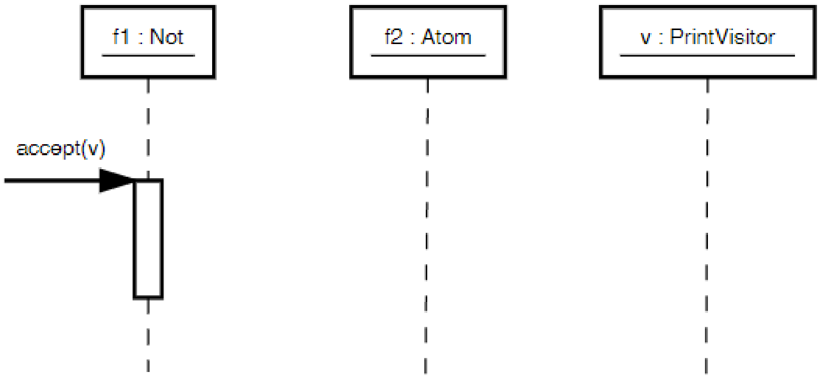


Figure 1: Template for the sequence diagram you have to draw.

Appendix E

Assignment Part 1

E.1 Provided code

```
public class EncryptionRefactor {

    public static void main(String[] args) {

    }

}
```

```
public class Encryptor {

    /**
     * A function that shifts a letter n places to the right
         (wrapping around and keeping its case)
     * @param c the given character
     * @param n the character should be shifted n places
     * @return the resulting character
     */
    private char shiftNplaces(char c, int n) {
        int result = (int) c;
        n = n % 26;
        int ascii = (int) c;
        if ((65 <= ascii && ascii <= (90 - n)) || (97 <= ascii &&
            ascii <= (122 - n))) {
            result = ascii + n;
        } else if (ascii > 90 - n && ascii <= 90) {
            int diff = 90 - ascii;
            result = 64 + n - diff;
        } else if (ascii > 122 - n && ascii <= 122) {
            int diff = 122 - ascii;
            result = 96 + n - diff;
        }
        return (char) result;
    }

    /**
     * Encrypts a string with the given encryption method
     * @param s the given string
     * @param em the given encryption method
     * @return the encrypted string
     */
    public String encrypt(String s, EncryptionMethod em) {
        StringBuilder result = new StringBuilder();
        switch (em) {
            case CAESAR3:
                for(int i = 0; i < s.length(); i++) {
                    result.append(shiftNplaces(s.charAt(i), 3));
                }
                break;
            case ROT13:
                for(int i = 0; i < s.length(); i++) {
                    result.append(shiftNplaces(s.charAt(i), 13));
                }
                break;
            case MIRROR:
                for(int i = s.length() - 1; i >= 0; i--) {
                    result.append(s.charAt(i));
                }
                break;
        }
        return result.toString();
    }

}
```

```
public enum EncryptionMethod {
  CAESAR3,
  ROT13,
  MIRROR
}
```

```
/**
 * The Factory to easily execute function for the test cases
 * You should edit these such that these function work with your
     refactored code
 */
public class EncryptionFactory {
    public static String encryptCaesar3 (String s) {
        Encryptor e = new Encryptor();
        return e.encrypt(s, EncryptionMethod.CAESAR3);
```

```java
    }

    public static String encryptROT13 (String s) {
        Encryptor e = new Encryptor();
        return e.encrypt(s, EncryptionMethod.ROT13);
    }

    public static String encryptMirror(String s) {
        Encryptor e = new Encryptor();
        return e.encrypt(s, EncryptionMethod.MIRROR);
    }
}
```

E.2 Solution I

```java
public class EncryptionRefactor {

    public static void main(String[] args) {

    }

}
```

```java
public class Encryptor {
    private EncryptionMethod em;

    public Encryptor(EncryptionMethod em) {
        this.em = em;
    }

    /**
     * Encrypts a string with the encryption method stored in em
     * @param s the given string
     * @return the encrypted string
     */
    public String encrypt(String s) {
        return em.encrypt(s);
    }

}
```

```java
public interface EncryptionMethod {
    public String encrypt(String s);
}
```

```
public class Caesar3 implements EncryptionMethod {

    /**
     * A function that shifts a letter n places to the right
         (wrapping around and keeping its case)
     * @param c the given character
     * @param n the character should be shifted n places
     * @return the resulting character
     */
    private char shiftNplaces(char c, int n) {
        int result = (int) c;
        n = n % 26;
        int ascii = (int) c;
        if ((65 <= ascii && ascii <= (90 - n)) || (97 <= ascii &&
            ascii <= (122 - n))) {
            result = ascii + n;
        } else if (ascii > 90 - n && ascii <= 90) {
            int diff = 90 - ascii;
            result = 64 + n - diff;
        } else if (ascii > 122 - n && ascii <= 122) {
            int diff = 122 - ascii;
            result = 96 + n - diff;
        }
        return (char) result;
    }

    @Override
    public String encrypt(String s) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < s.length(); i++) {
                    result.append(shiftNplaces(s.charAt(i), 3));
        }
        return result.toString();
    }
}
```

```
public class ROT13 implements EncryptionMethod {
    /**
     * A function that shifts a letter n places to the right
         (wrapping around and keeping its case)
     * @param c the given character
     * @param n the character should be shifted n places
     * @return the resulting character
     */
    private char shiftNplaces(char c, int n) {
        int result = (int) c;
        n = n % 26;
        int ascii = (int) c;
        if ((65 <= ascii && ascii <= (90 - n)) || (97 <= ascii &&
            ascii <= (122 - n))) {
            result = ascii + n;
        } else if (ascii > 90 - n && ascii <= 90) {
            int diff = 90 - ascii;
            result = 64 + n - diff;
        } else if (ascii > 122 - n && ascii <= 122) {
            int diff = 122 - ascii;
            result = 96 + n - diff;
        }
        return (char) result;
    }

    @Override
    public String encrypt(String s) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < s.length(); i++) {
                    result.append(shiftNplaces(s.charAt(i), 13));
        }
        return result.toString();
    }
}
```

```
public class Mirror implements EncryptionMethod {
    @Override
    public String encrypt(String s) {
        StringBuilder result = new StringBuilder();
        for(int i = s.length() - 1; i >= 0; i--) {
                    result.append(s.charAt(i));
        }
        return result.toString();
    }
}
```

```
/**
 * The Factory to easily execute function for the test cases
 * You should edit these such that these function work with your
     refactored code
 */
public class EncryptionFactory {
    public static String encryptCaesar3 (String s) {
        Encryptor e = new Encryptor(new Caesar3());
        return e.encrypt(s);
    }

    public static String encryptROT13 (String s) {
        Encryptor e = new Encryptor(new ROT13());
        return e.encrypt(s);
    }

    public static String encryptMirror(String s) {
        Encryptor e = new Encryptor(new Mirror());
        return e.encrypt(s);
    }
}
```

E.3 Solution II

All the files are the same as Solution I, except that `ROT13.java` and `Caesar3.java` are combined in a new, more general class `Shifter.java`. As a result, `EncryptionFactory.java` is also changed a little bit.

```
public class Shifter implements EncryptionMethod {
    private int shiftNumber;

    public Shifter(int shiftNumber) {
        this.shiftNumber = shiftNumber;
    }

    /**
     * A function that shifts a letter n places to the right
         (wrapping around and keeping its case)
     * @param c the given character
     * @param n the character should be shifted n places
     * @return the resulting character
     */
    private char shiftNplaces(char c, int n) {
        int result = (int) c;
        n = n % 26;
        int ascii = (int) c;
        if ((65 <= ascii && ascii <= (90 - n)) || (97 <= ascii &&
            ascii <= (122 - n))) {
            result = ascii + n;
        } else if (ascii > 90 - n && ascii <= 90) {
            int diff = 90 - ascii;
            result = 64 + n - diff;
        } else if (ascii > 122 - n && ascii <= 122) {
            int diff = 122 - ascii;
            result = 96 + n - diff;
        }
        return (char) result;
    }

    @Override
    public String encrypt(String s) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < s.length(); i++) {
                    result.append(shiftNplaces(s.charAt(i),
                        shiftNumber));
        }
        return result.toString();
    }
}
```

```
/**
 * The Factory to easily execute function for the test cases
 * You should edit these such that these function work with your
     refactored code
 */
public class EncryptionFactory {
    public static String encryptCaesar3 (String s) {
        Encryptor e = new Encryptor(new Shifter(3));
        return e.encrypt(s);
    }

    public static String encryptROT13 (String s) {
        Encryptor e = new Encryptor(new Shifter(13));
        return e.encrypt(s);
    }

    public static String encryptMirror(String s) {
        Encryptor e = new Encryptor(new Mirror());
        return e.encrypt(s);
    }
}
```

Appendix F

Assignment Part 2

F.1 Provided code

```java
public interface Formula {
}
```

```java
import java.util.Map;

public class FormulaFactory {

   public static Formula atom(String atomId) {
      return null;
   }

   public static Formula and(Formula leftOp, Formula rightOp) {
      return null;
   }

   public static Formula or(Formula leftOp, Formula rightOp) {
      return null;
   }

   public static Formula implies(Formula leftOp, Formula rightOp) {
      return null;
   }

   public static Formula not(Formula notOp) {
      return null;
   }

   public static final Formula TRUE = null;

   public static final Formula FALSE = null;
```

```
    public static String prettyPrint(Formula f) {
        return "";
    }

    public static Boolean evaluate(Formula f, Map<String,Boolean>
            env) {
        return null;
    }
}
```

F.2 Solution

```
public interface Formula {
    public int getPrecedence();
    public <Result> Result accept(FormulaVisitor<Result> visitor);
}
```

```
public class Constant implements Formula {
    private boolean value;

    public Constant(boolean value) {
        this.value = value;
    }

    public boolean getValue() {
        return value;
    }

    @Override
    public <Result> Result accept(FormulaVisitor<Result> v) {
        return v.visit(this);
    }

    @Override
    public int getPrecedence() {
        return 4;
    }
}
```

```
public class Atom implements Formula {
    private String name;

    public Atom(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    @Override
    public <Result> Result accept(FormulaVisitor<Result> v) {
        return v.visit(this);
    }

    @Override
    public int getPrecedence() {
        return 4;
    }
}
```

```
public class Not implements Formula {
    private Formula operand;

    public Not(Formula oper) {
        this.operand = oper;
    }

    public Formula getOperand() {
        return operand;
    }

    @Override
    public int getPrecedence() {
        return 3;
    }

    @Override
    public <Result> Result accept(FormulaVisitor<Result> v) {
        return v.visit(this);
    }
}
```

```
public class BinaryOperator implements Formula {
    private Formula left;
    private Formula right;
    private BinOp operator;

    public BinaryOperator(Formula left, Formula right, BinOp
        operator) {
        this.left = left;
        this.right = right;
        this.operator = operator;
    }

    public Formula getLeft() {
        return left;
    }

    public Formula getRight() {
        return right;
    }

    public BinOp getOperator() {
        return operator;
    }

    @Override
    public int getPrecedence() {
        return operator.getPrecedence();
    }

    @Override
    public <Result> Result accept(FormulaVisitor<Result> v) {
        return v.visit(this);
    }
}
```

```
public enum BinOp implements BinaryOperator<Boolean> {
    AND {
        @Override
        public Boolean apply(Boolean l, Boolean r) { return l && r; }
        @Override
        public String getRepresentation() {return "/\\";}
        @Override
        public int getPrecedence() {return 2;}
    },
    OR {
        @Override
        public Boolean apply(Boolean l, Boolean r) { return l || r; }
        @Override
        public String getRepresentation() {return "\\/";}
        @Override
        public int getPrecedence() {return 1;}
    },
    IMPLIES {
        @Override
        public Boolean apply(Boolean l, Boolean r) { return !l || r;
            }
        @Override
        public String getRepresentation() {return "=>";}
        @Override
        public int getPrecedence() {return 0;}
    };

    public abstract String getRepresentation();
    public abstract int getPrecedence();
}
```

```
public interface FormulaVisitor <Result> {
    Result visit(Atom form);
    Result visit(Not form);
    Result visit(Constant form);
    Result visit(BinaryOperator form);
}
```

```
import java.util.Map;

public class EvaluateVisitor implements FormulaVisitor<Boolean> {

    private Map<String,Boolean> env;

    public EvaluateVisitor(Map<String,Boolean> env) {
        this.env = env;
    }

    @Override
    public Boolean visit(Atom form) {
        return env.get(form.getName());
    }

    @Override
    public Boolean visit(Not form) {
        return !form.getOperand().accept(this);
    }

    @Override
    public Boolean visit(Constant form) {
        return form.getValue();
    }

    @Override
    public Boolean visit(BinaryOperator form) {
        boolean resLeft = form.getLeft().accept(this);
        boolean resRight = form.getRight().accept(this);
        return form.getOperator().apply(resLeft, resRight);
    }

}
```

```
public class ShowVisitor implements FormulaVisitor<String> {

    @Override
    public String visit(Atom form) {
        return form.getName();
    }

    @Override
    public String visit(Not form) {
        String result = "!";
        if (form.getPrecedence() >=
            form.getOperand().getPrecedence()) {
            result += "(";
        }
        result += form.getOperand().accept(this);
        if (form.getPrecedence() >=
            form.getOperand().getPrecedence()) {
            result += ")";
        }
        return result;
    }

    @Override
    public String visit(Constant form) {
        if (form.getValue()) {
            return "True";
        } else {
            return "False";
        }
    }

    @Override
    public String visit(BinaryOperator form) {
        String result = "";
        if (form.getPrecedence() >= form.getLeft().getPrecedence()) {
            result += "(";
        }
        result += form.getLeft().accept(this);
        if (form.getPrecedence() >= form.getLeft().getPrecedence()) {
            result += ")";
        }
        result += form.getOperator().getRepresentation();
        if (form.getPrecedence() >= form.getRight().getPrecedence())
            {
            result += "(";
        }
        result += form.getRight().accept(this);
        if (form.getPrecedence() >= form.getRight().getPrecedence())
            {
            result += ")";
        }
        return result;
    }

}
```

```
import java.util.Map;

public class FormulaFactory {

   public static Formula atom(String atomId) {
      return new Atom(atomId);
   }

   public static Formula and(Formula leftOp, Formula rightOp) {
      return new BinaryOperator(leftOp, rightOp, BinOp.AND);
   }

   public static Formula or(Formula leftOp, Formula rightOp) {
      return new BinaryOperator(leftOp, rightOp, BinOp.OR);
   }

   public static Formula implies(Formula leftOp, Formula rightOp) {
      return new BinaryOperator(leftOp, rightOp, BinOp.IMPLIES);
   }

   public static Formula not(Formula notOp) {
      return new Not(notOp);
   }

   public static final Formula TRUE = new Constant(true);

   public static final Formula FALSE = new Constant(false);

   public static String prettyPrint(Formula f) {
              ShowVisitor v = new ShowVisitor();
      return f.accept(v);
   }

   public static Boolean evaluate(Formula f, Map<String,Boolean>
       env) {
      EvaluateVisitor v = new EvaluateVisitor(env);
              return f.accept(v);
   }
}
```

Appendix G

Exam question & answer

Object Oriented Programming (NWI-IPI005)

Friday, 24 June 2022, 12:45-15:45

This exam consists of 4 exercises. The questions have different weights; the weight of each part is indicated in the margin. Check your answers carefully. The exam is closed book: You are NOT allowed to use a calculator, a mobile phone, or any printed material with the exception of your personal cheat sheet. Answer questions in terms of Java code.

1 Visitor pattern **total 13 points**

In mathematics, a polynomial is an expression consisting of a sum of terms, where every term has the form cx^n. Coefficients c are nonzero real numbers. Exponents n are natural numbers. For example,

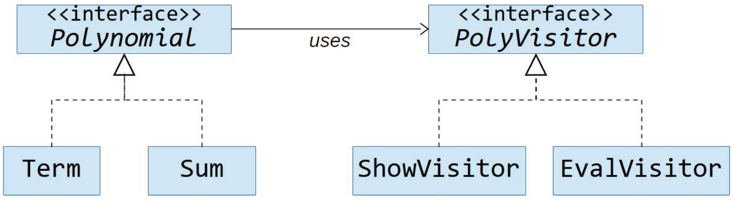
$$3x^4 - 4x^3 + 2x - 8$$

is a polynomial. In principle, polynomials can contain multiple variables, but in this assignment we only consider polynomials with a single variable x.

The representation we are going to use is based on the following recursive definition. A polynomial is:

- a *term* consisting of a coefficient and an exponent, or
- a *sum* of two polynomials.

We are going to introduce two operations on polynomials using the visitor pattern. As a basis for this we use the interface `Polynomial` implemented by the `Term` and `Sum` classes. The envisioned class diagram of the complete program is:



The interfaces `Polynomial` and `PolyVisitor` are defined as follows.

```
public interface Polynomial {
  <T> T accept( PolyVisitor<T> visitor );
}

public interface PolyVisitor<T> {
  T visit( Term t );
  T visit( Sum s );
}
```

The `PolyVisitor` interface has two implementations: `ShowVisitor` and `EvalVisitor`.

1.a) Define class `Term`. Since we only have 1 variable, it is not necessary to store this variable. Don't forget to implement the interface. Getters and setters may be omitted.

Solution:

```
public class Term implements Polynomial {
  private final double coef;
  private final int    expo;

  public Term(double coef, int expo) {
    this.coef = coef;
    this.expo = expo;
  }
  @Override
  public <T> T accept(PolyVisitor<T> visitor) {
    return visitor.visit(this);
  }
}
```

A) 1 bonus

1.b) Define class `Sum`. Again, getters and setters may be omitted.

Solution:

```
public class Sum implements Polynomial {
  private final Polynomial left, right;

  public Sum(Polynomial left, Polynomial right) {
    this.left = left;
    this.right = right;
  }

  @Override
  public <T> T accept(PolyVisitor<T> visitor) {
    return visitor.visit(this);
  }
}
```

A) 1 bonus

1.c) Implement class `ShowVistor` which should yield a string representation of a given polynomial. Here terms should be displayed as follows: If the exponent is 0, only the coefficient has to be given (i.e. no `x` or exponent) and if the exponent is 1, it should be omitted. For all other terms, both the coefficient and the exponent are shown. The example polynomial is represented by the string `"3,000000x4 + -4,000000x3 + 2.0x + -8.0"`. You can assume here that the getters from the above classes are available.

Solution:

```
public class ShowVisitor implements PolyVisitor<String>{
  @Override
  public String visit(Term t) {
    if ( t.getExpo() == 0 ) {
      return String.valueOf(t.getCoef());
    } else if ( t.getExpo() == 1 ) {
      return t.getCoef() + "x";
    } else {
      return String.format("%fx%d", t.getCoef(), t.getExpo());
    }
  }

  @Override
  public String visit(Sum s) {
    return s.getLeft().accept(this) + " + " + s.getRight().accept(
       this);
  }

  @Override
  public String visit(Product p) {
    return p.getLeft().accept(this) + "*" + p.getRight().accept(
       this);
  }
}
```

A) 1 bonus

1.d) Implement class `EvalVisitor` that evaluates a polynomial for a given value of *x*. The value of *x* should be added as a field to the class and initialized in the constructor that takes this value as an argument. E.g. If we take 3 as value for *x*, the above polynomial will evaluate to 133

Solution:

```
public class EvalVisitor implements PolyVisitor<Double>{
  private final double x;

  public EvalVisitor(double x) {
    this.x = x;
  }

  @Override
  public Double visit(Term t) {
    return t.getCoef() * Math.pow(x, t.getExpo());
  }

  @Override
  public Double visit(Sum s) {
    return s.getLeft().accept(this) + s.getRight().accept(this);
  }
```

```
    @Override
    public Double visit(Product p) {
      return p.getLeft().accept(this) * p.getRight().accept(this);
    }
  }
```

A) 1 bonus

1.e) Implement a class `PolyMain` in which the polynomial $4x^3 + 2x - 8$ is created, then printed on the screen and finally evaluated for $x = 3$.

Solution:

```
public class PolynomialMain {
  public static void main(String[] args) {
    Polynomial p3 = sum(term(4, 3), sum(term(2, 1), term(-8, 0)));
    ShowVisitor sv = new ShowVisitor();
    System.out.println(p2.accept(sv));
    EvalVisitor ev = new EvalVisitor(3);
    System.out.println(p2.accept(ev));
  }
}
```

A) 1 bonus

2 Streams **total 12 points**

We are requested to perform some data analysis on behalf of a popular video on-demand service. They want some insight into what is being watched and why. We are given the following interfaces:

```
public interface Actor {
  public String getName();
  public int getAge();
}

public interface TVSeries {
  public String getTitle();
}

public interface TVEpisode {
  public TVSeries getSeries();
  public int getDaysReleased();
  public Set<Actor> getAppearingActors();
}
```

We are additionally provided a method

```
public Stream<TVEpisode> getWatchedEpisodes();
```

Appendix H

Questionnaire after lecture

Questionnaire after lecture Design Patterns

Thank you very much for filling in this questionnaire after watching the lecture or reading the slides! Your answers are anonymous and are used for my research. Please fill this questionnaire in honestly, so we can use your feedback for better and more effective lectures! Thank you!

1. What are you studying?

Markeer slechts één ovaal.

○ Computing Science

○ AI

○ Anders: ______________________

2. How did you follow this lecture?

Markeer slechts één ovaal.

○ Being live present at the lecture

○ Watching the livestream live

○ Watching the recording back on Brightspace

○ Only looking at the slides of this week

○ Anders: ______________________

3. How did you experience the explanations of the design patterns? (use of UMLs, introduction with example, fully implemented example, too abstract, too complex etc)

4. How did you experience the used UMLs in the slides? (clear, too abstract, too complicated, not enough experience with UMLs etc)

5. Could you focus this lecture?

Markeer slechts één ovaal.

○ I could easily focus the whole lecture
○ I could easily focus most of the lecture
○ I could focus around 50% of the lecture
○ I could focus a small part of the lecture
○ I was very difficult to keep the focus in the lecture
○ Anders: ____________

6. How did you experience the exercises in the lecture?

7. How did you experience the programs presented in Netbeans instead of code snippets on the slides?

8. How did you experience this lecture compared to the other (Object Oriented Programming) lectures? And why?

9. Other comments/remarks/?

Appendix I

Questionnaire after assignment

Questionnaire after assignment

Thank you very much for filling in this questionnaire after making the assignment about design patterns! Your answers are anonymous and are used for my research. Please fill this questionnaire in honestly, so we can use your feedback for better and more effective lectures! Thank you!

1. What are you studying?

   Markeer slechts één ovaal.

   ○ Computing Science

   ○ AI

   ○ Anders: ______________________________

2. How did you experience the difficulty of the assignment?

   Markeer slechts één ovaal.

   ○ Too difficult

   ○ Challenging

   ○ Good

   ○ A bit easy

   ○ Too easy

   ○ Anders: ______________________________

3. Which problems did you encounter in the first part of the assignment (the refactor exercise). How did you solve these problems?

   ____

   ____

   ____

   ____

   ____

4. Which problems did you encounter in the second part of the assignment while constructing the UML? How did you solve these problems?

5. Which problems did you encounter in the second part of the assignment while implementing the formulas (the visitables)? How did you solve these problems?

6. Which problems did you encounter in the second part of the assignment while implementing the visitors (pretty printer & evaluator)? How did you solve these problems?

7. Did you encounter any other problems?

8. Were the slides useful for this assignment? Was there a clear connection between the taught material and the assignment?

9. Other comments/remarks/?

Appendix J

Codes for the interviews in the practical sessions

- Part 1 of the assignment
  - Not finished
  - No problems
  - Only minor problems
  - Difficult to recognise the design pattern
  - Difficult to implement the design pattern
  - Solved by asking TAs
  - Solved by looking at the slides
  - Solved by looking at the internet
  - ...
- Part 2 of the assignment
  - Not finished
  - No problems
  - Only minor problems
  - Problems with constructing the UML
  - Problems with the Visitors
  - Problems with the Visitables
  - Solved by asking TAs
  - Solved by looking at the slides
  - Solved by looking at the internet
  - ...

- Lecture
  - Clearer
  - Exercises were helpful
  - Live codings sessions were helpful
  - ...
- Tutorial session
  - Tutorial slides were useful
  - ...
- General Remarks
  - ...

When students encountered problems in the assignment which were so small or minor, we label it as a general label "Only minor problems". If there are more specific mistakes that students made in the exercise, we created a new, more specific code for it, as long as there were multiple students that made that specific mistake.

Appendix K

Rubric Exam Results

1.a) Define Class Term and 1.b) Define Class Sum (Visitables)	1.c) Implement Class ShowVisitor and 1.d) Implement Class EvalVisitor (Visitor)	1.e) Implement Class PolyMain (Using the Visitor Pattern)
Correct; The application of the Visitor pattern was correct, thus using an accept function with a correct call to the visit function	Correct; The application of the Visitor pattern was correct in the Visitors	Correct; The function calls to use the Visitors are correct, e.g. poly.accept(new PrintVisitor())
Visitor call incorrect; There was a minor mistake in the accept function. This category is distinguished further depending on the results	Visit function's body is incorrect; There was a minor mistake in the visit functions. This category is distinguished further depending on the results	Function calls are incorrect; There was a minor mistake in the function calls. This category is distinguised further depending on the results
Accept function completely incorrect; An accept function was defined, but the body was empty or no elements of the Visitor pattern could be recognised	Visit functions are completely incorrect; A visit function was defined, but the body was empty or no elements of the Visitor pattern could be recognised	Function calls are completely incorrect; There were function calls in the code, but they were too different than the solution to be used
Accept function missing; The class was specified, but no accept function was included	Visit functions are missing; The Class was specified, but no visit functions were included	
Completely incorrect; The entry was blanc or the code was not useful	Completely incorrect; The entry was blanc or the code was not useful	Completely incorrect; The entry was blanc or the code was not useful

Appendix L

Online Quiz results

	I	I	I	I	UML	AE	AE	AD	UML	AD	AE	AD	AE	AD	AE	
	Introduction				Strategy				Decorator				Visitor			
									After break							
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	%
Student 1	C	D	B	A	A	A	B	B	A	A	A	C	C	C	C	0,53
Student 2	A	C	B	C	D	A	B	B	B	B	A	A	A	C	A	0,67
Student 3	-	-	-	-	-	C	B	B	A	A	A	C	A	C	A	0,8
Student 4	A	C	B	C	A	A	B	A	C	A	A	C	A	A	A	0,73
Student 5	A	C	B	C	B	A	-	B	A	C	-	B	A	A	-	0,67
Student 6	-	-	-	-	-	-	-	-	A	A	A	C	-	-	-	0,75
Student 7	A	C	B	C	D	C	B	A	B	-	-	-	-	-	-	0,67
Student 8	A	C	B	C	A	C	B	B	A	C	A	C	A	C	C	0,87
Student 9	-	-	-	-	D	-	-	-	-	-	-	-	-	-	-	0
Student 10	-	-	-	-	-	A	B	B	A	A	A	A	A	C	A	0,7
Student 11	A	C	B	A	-	A	B	-	-	A	A	C	-	-	-	0,78
Student 12	-	-	-	C	D	A	B	C	-	-	-	-	-	-	-	0,8
Student 13	A	C	B	C	A	A	B	C	A	C	-	-	-	-	-	1
Student 14	A	C	B	C	A	C	B	A	A	A	A	C	A	C	A	0,87
Student 15	A	C	B	-	-	-	B	A	-	-	-	-	-	-	-	0,8
Student 16	A	C	A	D	A	-	C	A	A	A	A	C	A	C	A	0,64
Student 17	A	C	B	A	A	A	B	B	A	C	A	C	A	A	-	0,79
Student 18	A	C	B	C	A	A	B	B	A	B	A	C	B	A	A	0,73
Student 19	-	-	-	-	A	A	B	B	-	C	A	C	-	-	-	0,86
Student 20	A	C	B	C	A	A	B	B	A	A	A	C	A	C	C	0,8
Student 21	A	C	B	B	A	A	B	A	A	C	A	C	A	A	A	0,8
Student 22	A	C	B	A	D	A	B	B	-	-	-	-	-	-	-	0,63
Student 23	A	C	C	C	D	C	B	A	A	C	A	C	A	C	A	0,8
Student 24	-	-	-	C	-	-	-	-	-	-	-	-	-	-	-	1
Student 25	-	-	-	C	-	-	B	B	-	-	-	-	-	-	-	0,67
Student 26	-	-	B	C	-	-	-	-	-	-	-	-	-	-	-	1
Student 27	A	C	B	C	C	A	B	A	B	A	-	A	-	-	-	0,55
Student 28	A	C	B	C	A	A	B	B	A	C	A	C	A	A	A	0,87
Student 29	A	D	B	C	-	-	-	-	-	-	-	-	-	-	-	0,75
Student 30	B	-	C	C	-	-	-	-	-	-	-	-	-	-	-	0,33
Student 31	-	-	-	-	-	-	-	-	A	C	A	C	-	-	-	1
Student 32	-	-	-	-	-	-	-	-	-	-	-	-	A	A	B	0,33

Student 33	B	C	B	C	-	-	-	-	-	-	-	-	-	-	-	0,75
Student 34	A	C	B	C	D	A	B	C	-	A	A	-	A	-	A	0,83
Student 35	A	C	C	A	A	C	B	B	A	A	A	C	A	A	A	0,67
Student 36	A	C	D	C	A	C	C	B	-	C	A	-	-	-	-	0,7
Student 37	A	D	B	C	D	A	B	C	B	C	A	-	-	-	-	0,73
Student 38	-	C	B	C	C	A	B	B	B	B	A	C	A	C	B	0,64
Student 39	A	D	B	C	A	A	C	B	A	A	A	C	A	C	A	0,73
Student 40	A	C	B	C	-	-	-	-	-	-	-	-	-	-	-	1
Student 41	A	D	B	A	C	A	B	B	C	C	A	C	A	C	A	0,67
Student 42	-	-	-	-	-	-	B	A	A	B	A	C	B	-	A	0,63
Student 43	-	-	-	-	-	-	-	-	A	C	A	C	A	-	-	1
Student 44	A	C	B	C	A	C	-	-	-	-	-	-	-	-	-	1
Student 45	A	C	B	C	-	-	-	-	-	-	-	-	-	-	-	1
Student 46	A	C	B	D	A	A	C	B	A	C	A	C	A	A	A	0,73
Student 47	C	C	D	A	A	C	B	C	-	-	-	-	-	-	-	0,63
Student 48	-	-	-	-	B	A	B	B	D	A	A	A	A	C	A	0,55
Student 49	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Student 50	-	-	-	-	A	A	B	C	-	-	-	-	-	C	A	1
Student 51	A	C	A	A	A	C	B	B	A	C	A	C	-	-	-	0,75
Student 52	D	C	D	C	D	A	C	A	A	C	A	C	A	A	A	0,6
Student 53	-	-	-	-	A	A	B	B	A	C	A	C	A	C	A	0,91
Student 54	A	C	B	C	B	C	B	A	B	A	A	C	B	C	-	0,64
Student 55	-	-	-	-	-	A	B	C	A	A	A	C	A	A	A	0,8
Percentage correct	0,9	0,9	0,8	0,7	0,6	1	0,9	0,2	0,7	0,5	1	0,8	0,9	0,6	0,8	
Total answers	36	36	38	40	36	38	40	40	34	37	34	33	29	27	26	
Correct answers	31	31	30	29	21	38	35	7	25	20	34	28	25	16	21	