COMP47480 Contemporary Software Development

Lab Journal

Dawid Skraba (19433692)

BSc. (Hons.) in Computer Science

UCD School of Computer Science

University College Dublin

May 2023

Table of Contents

1 Agile Methods

1.1 Work Done

During this practical in our groups, we brainstormed as many ideas as we could that we thought would make a software project successful. We thought of a wide range of reasons, from having good team chemistry to planning. Then we select eight key criteria we thought were the most important. Finally, we then got a sheet with the twelve agile principles. We went through these and marked which our eight criteria covered at least one of these principles. Overall, we covered all but one principal.

1.2 Reflections

This exercise was very relevant to my experiences in the industry. I have completed a six month internship in third year and worked on a team which used the agile methodology. During my time with the team I learned how a team works under these principles. This experience also helped me understand what makes a successful software project. I got to experience first-hand what works and what doesn't, I think that's why we could brainstorm quite a few ideas during the exercise. My teammate also had experience in the industry so we exchanged our ideas and discussed them. This highlighted to me how different companies/teams work together. During my time in the company it was engrained in me that communication, testing and having good team chemistry and absolutely crucial. We would have regular team-bonding exercises and we were constantly encouraged to engage with each other and exchange ideas. Another idea I came up with is that testing everything multiple times is crucial and anything we produce must be verified by your teammates before being sent out to production. I think that these principles came from me rather than my teammate is that these ideas are more important in the telecommunications industry as one small mistake could have drastic consequences to the companies entire network. I felt that my teammates experience was much more relaxed on these and was more inclined to work on something by themselves too. This was very interesting to us, and this difference between us, I think forced a great diversity of ideas.

However, when we had to match our ideas to the twelve agile principles we covered all but one. After discussing why, we understood that this was one of the similarities between our experiences. Our diversity in experience worked to our strength here, as we both worked on agile teams, that worked in their own small different ways, but we both had one thing in common. We came into our teams in the middle of their projects, we were not there from the start and were dropped straight into development work. I think this is way we didn't think that "The best architectures, requirements and designs emerge from self-organising teams" was a principle of an agile working environment. We didn't experience teams self-organising and delegating work to one another, so neither of us thought that this was part of a successful team as we simply didn't get to experiences this in teams that followed the agile methodology.

This to me was very interesting, so I did some reading as to why this was principle lead to a successful team. I thought that this would be beneficial to me as I didn't really understand yet why this is needed in the agile process. I hoped that this would maybe help me in the future form better teams that will lead to software projects running on budget and on time(a very hard task)! From my understanding the main benefit is that each person on the team wears many hats, and so gains competency in many areas of software development. A team can rotate roles by each task and so each member of the team fully understands the full scope of the project as they have worked on multiple different aspects of it. Another benefit I would have liked to see during my time in the internship would be self-delegating tasks among each other. I feel that this way everyone would pick the task they are most comfortable with and so each task is completed to the highest possible standard. My team had a scrum master who mostly gave tasks to the team. Some feedback was considered if someone didn't feel comfortable with a task, but mostly the scrum master handed tasks out. I think that letting teams to this by themselves leads to much better results.

2 Modelling with UML

2.1 Work Done

In this practical we firstly modelled a covid tracker system using a use case diagram. This involved understanding all the parties involved and the uses of the system they will be able to use. In exercise two we wrote out what the given class model permits and explicitly disallows, by examining the relationships these objects had. Then we created a high-level class diagram for the covid tracker below and lastly, we drew out a sequence diagram. We modelled both parties (GP and user) and modelled their behaviour with the systems methods.

2.2 Reflections

During my studies in University, I must admit that I drew out UML diagrams before implementing code, only a handful of times. However, I think that planning out how you're going to tackle a certain problem is incredibly useful and from experience I can confirm that it reduces the number of system-wide errors. I think that if UML diagrams were more widely used it would reduce the number of projects that run over budget and would help them be delivered on time. This is a massive problem in the industry as on average 45 percent of projects go over budget and 7 percent on time, while also delivering 56 percent less value[1]. As we can see, we need new techniques in development, but why aren't we utilising UML diagrams. I set out to understand this.

I came across a very interesting article which tries to answer this question[2]. Firstly, it's worth mentioning the advantages of UML. It's the most used and flexible tool for drawing out software relationships in a system. Even though it is rarely utilised most developers know how to use it effectively. It can be used for a wide variety of software and not just object-oriented designs; it has a wide effect on all software systems. It's also relatively easy to use and learn as its language is not complicated and most of the details are based on common sense thinking. Some negatives that are outlined are that its notation is not necessary to communicate the software system to its developers. Developers are not fond of adhering to a set of rules and would much rather draw their software implementation out informally. Some companies deem this unnecessary for their way of work as they feel that development needs to be minimalistic and so sees no benefit in complex and expansive UML diagrams explaining their system. Lastly the article states that "Absence of design documentation is fine in the short run, but it can become a problem in the long run when you need to communicate the design to a developer who is in another country, or someone who will be joining the team six months later. UML becomes a huge help in such circumstances and alleviates ambiguity and questions regarding the design." I found that companies do deem this software methodology useful but are hesitant for the reasons stated above and the fact that most developers simply don't like planning and drawing out their ideas, they simply want to sit down and code. I think there is a disconnect with the developers and the business managers, who would like these diagrams to better track their progress and to easier understand the systems they are building.

I find this to also be the case as most of the time I think that drawing out my implementations of a problem, is a waste of time and that time would be better spent coding. However, in this practical I got reminded of how drawing out your software system makes you think about the big picture, rather than coding along and encountering problems as they come. I think that this sort of development is the best practice for complex systems and could also be useful for smaller systems too. I feel that this stigma to use diagrams being unnecessary is incredibly harmful and the statistics, don't lie, we need to change how we tackle large-scale software systems, and I think this way would help deliver projects on-time and on-budget.

3 Seminar

3.1 Summary

- How Amazon or similar companies develop software
    - No one size fits all, team specific
    - Move fast and break things, which leads to problems as company matures
    - Agile methodology used
    - Legacy is common and structure can be messy
    - Success comes from teams that are left to their own devices
    - Both meta and amazon, strong engineering principles, AWS more top-down approach
- How Software Development changed over time
    - Cloud Computing
    - Huge new possibilities, scaling
    - Over time, engineers lack core knowledge of how it works and just build on top of it
    - Interestingly, AWS did not use AWS products in development
    - Reliability Vital
- While designing was quality in mind
    - No, speed more important
    - No design reviews: understand requirements, plan, build
    - This changes over time -> more design and quality focused
    - Weight slows you down, as integration becomes an issue
    - Checklist manifesto used for quality and reliability of code
    - Now common to hire a reliability engineer, reliability reviews
    - Hard to keep this quality first mentality as company grows
    - Ownership ties into quality: when responsible engineers build reliable products
- Did The potential for outages change the code design?
    - Edge cases bound to occur as scale is huge
    - Will lead to changing your testing strategy
    - Incident reviews very crucial
- What development methodologies used
    - Solve problem how you see fit
    - Agile Scrum popular, Kanban with DevOps
    - Teams that plan well and take time to design do well
- To what extent can teams choose their languages or technologies?
    - Free to use anything within reason
    - Constrained as testing frameworks and tools built inhouse with specific languages
    - Pick language people are comfortable, when you need help

3.2 Reflections

The talk with Karl was incredibly interesting and I found the subjects that were covered to be thought provoking and intriguing. The first part of the talk I found to be interesting. Karol talked about how Amazon and other big companies develop software. I was very interested in this as I only had experience working for one Software company through my 3rd year internship. I worked for Ericsson which is a leader in the telecommunications field. My thinking was that big software companies like Amazon or Ericsson develop software in a similar way. Karol mentioned that Amazon put speed ahead of any other priorities and the mentality was to move fast and break things. Then you look back and fix things. This was defiantly not the experience I had working for Ericsson. There reliability and security were at the forefront of every software developer. Before jumping into coding straight away a new feature, meetings were held to plan the design

and to iron out any issues in the design stage. At Amazon this was not mandatory and only certain teams participated in this way of developing software. Karol mentioned that successful teams were the ones that planned and designed first. I think that the two different ways of designing and developing software comes with the industry that you are in. Telecommunications is a very important field and if there is a bug that flies under the radar then tens of thousands of people could be cut from the internet. This was being reminded at every company wide event. I found this way of developing to be reassuring as whatever I pushed, I had to get my team to sign off on it. This helped me develop better and cleaner code, when they sometimes would come back to me and show me a better way of writing a piece of code or some other advice was given. One thing that was common in both companies is the agile methodology, thus proving superior to any other as two vastly different companies swear by its effectiveness.

Another thing Karol mentioned was how ownership of the code you write determines the quality. At first, I wasn't quite sure how this is the case, but it does make sense. When a product is deemed to be your and your responsible for it, your way more likely to develop more reliably and more quality oriented code than before, because you don't want to get a call at two am, saying something broke, come fix it. I think there needs to be a balance here as I feel like people would become territorial and only work on their products, thus limiting collaboration. There needs to be collaboration for a team to succeed as sometimes others find better solutions than you or see something you haven't. I think a solution to this ownership model would be to gamify it. If you help fix a bug or propose a better change to someone's code you get some points awarded to you, thus incentivising collaboration. The benefit of code ownership would still be there but also collaboration between teams and people would still be upheld as people would help each other far more if they are compensated for it. I would like to work under such a system as it takes the positives from both methodologies. I worked under a team-ownership system, where products are not owned by single developers but teams, which has many benefits but only forces collaboration within teams. In my opinion the idea laid out above would work better. Karol also mentioned that teams that are "left alone" thrived, meaning not being micromanaged by managers. I found this to be true as my previous colleagues highly spoke about managers that left design and implementation decisions to them and only came in when problems arose. I do believe that some oversight is needed, but engineers are smart people that need to be trusted to make the decisions they feel are best. This also raises an employee's mental state, as they feel they are entrusted by their bosses and that they have confidence in their ability to carry out the work however they see fit.

4 Testing

4.1 Work Done

In this lab we wrote a simple program using test Driven Development. We first wrote test cases that failed, then wrote code so that these tests can pass and then we refactored the code to make it easier to read, and so on until we have a fully working program. We then tested the test coverage for the code written by writing some print statements and analysing the output. Then we worked on the second part of the Testing lab. I performed mutation testing to my code from the previous lab to find bugs in my code. Then I worked with mocking. I created a voterInterface in the BallotTest class to test the voter class and how its interacting with out ballot class.

4.2 Reflections

During this practical I had the opportunity to write code using the test driven development for the first time. At first I found it pretty difficult as it was simply not how I wrote code before. I was thinking of how to implement the classes and then thinking of the test classes after. This methodology is wrong, as its working backwards and it dissolves the full benefits of test driven development. For the first class implementation I was just trying to get used to writing tests first, without thinking of how I'm going to implement multiple methods. It took me a while to get comfortable with it, but when it came to writing the second class of 'Ballot' I understood the benefits of writing code this way and only thought of writing test classes before any implementations. I found it makes writing code easier, as it forces you to break up a problem into small chunk sized problems. You are forced to write simple test cases and then write code which passes this test, after a while, without realizing it you have a full working program. Writing code first, you are more prone to make simple mistakes which could later mean that your whole implementation needs a rewrite. Being forced to pass simple tests and then refactoring makes it so that the code works as intended at every step and that the code you write is of good quality.

However not many developers use TDD to write code in the industry and even a fewer amount conform to each step of the process. Its been surveyed that approximately 3.4% developers use TDD[4] and 33% of developers do not begin by writing a test case first[3], violating a crucial part of the process. The flaw with this technique seems to be that developers seem to be lazy and want to get to writing code as soon as possible. Another reason is that it's a big change for developers to undertake and they have to change how they work in a fundamental way, which can be difficult. A study conducted by Janzen and Saiden[5] found that developers saw benefits in this technique but found it too different from how they normally work. This leads to the low numbers of developers that actually practice TDD as we found above. Another study also found that 56% of developers had difficulty adopting the TDD process[6]. I concur with these studies as it was a drastic change from anything that I had done before and required a lot of effort to get out of thinking about implementation first. As hard it is to change your mindset, many studies prove the effectiveness of this process. One experiment found that a test-first approach compared to a test-last approach led to 50% more code written with similar complexity and also improved their confidence in the project as both groups were asked how confident they are in their project on a 1 to 5 scale and the test-first group averaged a 4.75 and the test-last group scored a measly 2.5[7]. After studying these figures it's clear that we might want to think harder about how we write our code. Before this practical, I didn't think much of different methodologies and schools of thought in this area. I just sat down and worked, I feel like that's how a majority of people code. At the very least this should be a catalyst to think harder about our ways of work, we should discuss these more often. Beyond this practical I haven't heard much about this area. This needs to change as it's crucial to our productivity and when Netflix is down again, maybe an engineer didn't follow the TDD methodology!

When testing the code coverage of my program I found that my branch coverage for my entire application was 43% and statement coverage was at 66% and path coverage was 51%. These numbers at first shocked

me and made me think that I applied TDD wrongly. However, I don't believe that was the issue. The reason for these relatively low scores is that I added functionality in my code where a vote can be random if users specifies and the voting has two methods for when votes are randomized and one where votes are specified when creating a voter. This is the reason my test coverage is around 50% as I couldn't test half my methods as they outputted a random vote. This was maybe not required to put into the implementation, but where it was possible to test my implementation my test coverage is very good and covers all possible paths. Looking back now, I wouldn't have added this functionality as it deteriorated my test-coverage. It's hard to test a function if the output is random. This thought me a good lesson, about how test-coverage is calculated and how in certain cases it may be impossible to get it to an adequate level, like in a function that produces a random output.

During mutation testing I went through my 'election' method in the Ballot class and changed certain statements around and ren tests to see if I could find a bug in my code. After looking for a while I found a bug where I check the amount of yes votes against the number of no votes. I didn't accommodate for when there is a draw in the vote. I noticed this by changing the greater than sign to greater than or equal to sign, then when I ran the test cases some failed. I went back to the code and fixed this bug. I found mutation testing a great way to find bugs in your code that you didn't think of and that your unit tests don't cover. In my case this worked perfectly as this way of testing helped me find a bug that I overlooked. After this exercise I will make sure to use this technique in my development.

The next exercise involved mocking. I mocked the Voter class in the ballot test class so that we could easily test how the voter class behaves and integrates with other classes. Updating my test classes to use the VoterMock class was easy and straightforward and only had to make a minimal change to the Ballot class so that it will accept voter interfaces. Mocking made the testing of the voter class streamlined and really easy. To test of the inform method is called by each voter I simply extended the voter mock class to update a counter when its informed of the election. The benefit of doing it this way is that my original voter class didn't have to change at all to test this functionality. I saw the benefit of mocking straight away as it made testing some functionalities, that normally would require workarounds like changing your original code just for testing purposes, a lot simpler. If you want to test some other voter class functionality, all you have to do is change the voter mock class. This is what I did to check if the vote function is called only by the voters who will still have an impact on the vote. This way of testing also helps developers test code that hasn't yet been developed. If I wanted to test the ballot class against some other class without yet implementing that class, I could simply mock it. This mock class would act as if that class has already been implemented. This made testing a lot more effective as we can fully test one class without having implementing all the classes that we call upon. It also makes it easier to isolate a class and test its own functionalities, therefore it's obvious where from our code the test case fails and not one of the dependencies[8]. Developers therefore often use mocking to help them test there code more efficiently. I found that using mocking from the start of developing this small program would make it easier to test the Ballot class. Since we didn't make use of mocking from the start we had to start developing the voter class as it didn't depend on any class, then develop the Ballot class which used the voter class. When mocking we can develop more complex classes first and mock simpler classes that are used in that class. I prefer this method of developing, better and in my experience makes programming more resilient to bugs and makes you think of things that you wouldn't until later on in development.

5 Principles

5.1 Work Done

During this lab we walked through multiple software principles and modified code which breaks some of these principles, in such a way that follows best practice. The principles that we looked at during this practical are: The Open Closed Principle (OCP), The Single Responsibility Principle (SRP), The Interface Segregation Principle (ISP)and The Law of Demeter.

5.2 Reflections

Starting with the Open Closed Principle, the obvious problem with the implementation is that postageStamp right now can only be one shape. In its constructor the object of Square is passed in, if we wanted to add another shape like Circle, we would need to change the postageStamp class and create a new constructor with the new shape. This is not ideal and violates the OCP principle as this class needs to know about the shape its receiving. What I did to fix it, is to abstract that closed-knitted implementation and made the object passed to the postageStamp class be a Shape object. Shape is an interface that includes methods that all shapes need to implement, in this case just the toString method. We can create new shapes and simply implement the Shape interface. This way the postageStamp class doesn't know which type of shape its receiving, just that it's a shape, so it can perform the toString method. Now the postageStamp class is closed for modification, but open for extension. This means that when the class is closed for modification, the classes code should not be changed once it has been written. This is for the reason that modifying could cause bugs and could break functionality for the code that depends on it. When a class is open for extension, it means that it should be designed in such a way that its behaviour can be customized without needing to modify its code. We can accomplish this by inheritance, dependency injection or composition. After modifying this example, the other four principles of SOLID are adhered to. The Single Responsibility Principle, is followed as each class has exactly one responsibility and that responsibility is entirely capsulated by that class. The shape interface represents a shape, square and circle represent specific shapes and postageStamp represents a postage stamp by taking in a shape. The Liskov Substitution Principle states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program. The Shape interface acts as the base class and Square and Circle classes, can be used interchangeably without affecting the correctness in the program, as the postageStamp class works exactly the same whether those two classes are passed in, so this principle also holds. The Interface Segregation Principle (ISP) states that clients should not implement interfaces they don't need to use. So we can clearly see this code also follows this principle as both classes use the shape interface and make use of the one method it introduces the toString method. The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules but depend on abstractions. The postageStamp class depends on the Shape abstraction rather that the two specific implementations. So we can new shapes without any changes to that class, thus adhering to the DIP principle.

Then we fixed code that clearly violated the Single Responsibility Principle. Considering that a module attendee is permitted to attend a module several times, and so may have several grades for the module, the updated implementation would violate the SRP principle. This violates the principle as moduleAtendee class as it only handles one grade. To fix this problem I would introduce a HashMap for the grades of the module. The keys of the HashMap would be the instance of the attempt at that module and the value would be the grade. So now this code would adhere to the SRP principles.

The Interface Segregation principle is violated with this code as all clients use the same interface or class of bankAccount. This means that client has access to methods it should not have, like applyOverdraft and audit. The bankAccount class will not have the same functionality for all of its clients that have access to the bank Account class. In order to fix this I created two new interfaces ClerkInterface and CustomerInterface that

define methods of the bankAccount class that the clerk will be able to use and the Customer. Now the bankAccount class implements these interfaces. I changes the Clerk and customer classes as now they aren't passed in instances of bankAccount, but receive their own interface to the bank Account. This means now that they are limited to the methods they can perform on their bank Account. This means now that this code follows the Interface Segregation Principle as each customer has their own interface to a concrete class that they will use to perform action on that class.

Finally we looked at code that violates the law of Demeter. Which states that components should have limited knowledge of other components in the system. This is to reduce coupling between components and improve maintainability and adaptability of the code. The code provided violates this law as, the shopkeeper when charging the customer, gets the customer's wallet and subtracts money form their wallet. This is a clear violation of the law, as that's too much coupling in that class and also simply isn't how a shopkeeper and customer interact. The shopkeeper doesn't take the customer's wallet and take their cash, the customer does that himself, by receiving the information of how much they are charged. To fix this I changed the customer class so that they can subtract money from their wallet and they can look into their wallet and check their balance. So now the shopkeeper only interacts with the customer class, by asking their balance and then asking to subtract that amount from their wallet. This now means the code follows the law of Demeter as the two classes are not strongly couples and the shopkeeper class only interacts with the customer class, which then handles the subtraction of the money.

6 Refactoring

6.1 Work Done

(i) The statement method in the Customer class is too long and more complicated than it needs to be. The cost of the rental vehicle should not be calculated in the Customer class. This is beyond the scope of that class. This method should be broken up into multiple methods, for simplicity There are many comments in the statement method, this shows that the method is too complicated. A method that The getTitle method, in the Vehicle class is named incorrectly, as its returning a model object it should be named getModel The law of Demeter principle is broken in the statement method in the switch statement. In the test Class the expected string was misspelled, so the test cases firstly failed The vehicle types in the Vehicle class are static ints and should be Enums.

(ii) Both classes are acting like data classes. In order to fix these you should use the remove setting method on any field that should not be changed setVehicle. Check where we use getters and setter in the Customer class and bring that functionality up to the Vehicle or Rental classes.

(iii) The Customer statement should not figure out the cost of the rental, that should be handled by rental. The method is too long also. The method also has too many comments, meaning it's too complicated to understand by just reading the code. The switch statement could be changed to use polymorphism, so to use each of the vehicles as subclass. The Customer class is acting as a God class, as its performing all the functionality.

(iv) The Customer class should be refactored and broken up. It holds all the functionality of the entire program and the Vehicle and Rental classes act as Data classes and nothing else. Some of that functionality should be brought up to the Data classes so that the law of Demeter is no longer violated.

(v) The switch statement is performing all of the functionality for when the vehicle type is of a certain type. To reduce the complexity of this, we should use polymorphism. So create multiple sub-classes for the vehicles and move all the functionality from the switch statement to the subclasses. The Vehicle would be an abstract class and the subclasses would share some methods.

(vi) Yes, the code block calculates the amount needed to pay for the vehicle rented based on the type and days rented.

(vii) That should not be extracted as we will be updating this value multiple times. The for loop calculates all rentals so we will updating this value multiple times for all the rentals. So the value should not be extracted as it will be updated from the returning newAmount from the function.

(viii) The code does change one single variable, so it would be appropriate to extract this method.

(ix) The days rented should be passed to the new method as to calculate the amount correctly.

(x) The IDE's refactoring is not as effective as doing it by hand as it didn’t apply polymorphism to the vehicle sub-classes, it simply moved the switch statement to a new method, which does simplify the overall method but the new method still suffers from that code smell, which i corrected by adding the vehicles subclasses, also the functionality of calculating the amount

owed is still being calculated in Customer, which should not happen and I moved that to happen at each of the subclasses.

(xi) Trying out the extract method on other parts of the code, shows that it doesn't work that great. This is because when we select to extract a certain functionality, the scope of changes is in just that highlighted area, which is not ideal, as more structural changes are needed to make refactoring worthwhile e.g. switch statement. The freqRentalPoints extraction method is moved outside the statement method, but it's still in the Customer class, which this should be handled in the Rental class.

(xii) I prefer to keep the delegated method. In my opinion this is useful as this delegate method just forwards the method call to the new function and this might be useful if some clients use the "old" way of calling this method and this way will still work without changing how the client calls the new method. Another reason you might want to keep the delegated method is if the original class provides a high level abstraction, which you might want to keep. I however i removed the delegated method, as to not have code duplication, I have a reference to that new class from the old method, so clients using the old method, will still work, without having code duplication.

(xiii) After refactoring all my testcases passed like expected, so we succeeded in changing the location of the envy method to that class, without altering the code functionality. I would not add any new testcases now as the functionality is exactly the same as before.

(xiv) I renamed the method from "statement" to "generateStatement", as it's a more accurate description of what this method is trying to achieve. Comments are not needed here so i would delete them. The code here is fairly simple and simple to follow, so the developer will know what it's doing without the need to explain the code more. Having many comments often means the code you have written is too complicated and should be re-written, so that developers can glance at the code and understand what it's doing. Another reason to not include them (or to try avoid them) is that when code is updated, comments are often overlooked and then comments serve the opposite purpose, complicating the code more. Reading code should be enough to understand what it's doing, if that's not the case....Refactor!

6.2 Reflections

During this lab I realized that identifying code smells was only the beginning of the challenge. M. Fowler[9] defines code smells as "surface indications that usually correspond to deeper problems in the system". I read this as, when we encounter many code smells in a particular area of the code and even though these aren't critical bugs, they indicate that the code written was written so poorly and with bad programming standards. Code smells are a great way to check if your code may require a rewrite and suggest that other developers may not fully and easily understand the code you wrote. During this practical I came across a method that is considered a "rotten". This is because it's a long and complicated method. The fact that is long is easy to see, it also had many comments explaining what certain lines of code do. This to me didn't seem out of the ordinary, as I comment my code to explain certain functionalities that might be hard to understand. However in this practical this belief of mine was challenged. The use of comments in code is often seen as a best practice, as they provide essential context and clarification for developers who might later interact with the code[10]. In some cases, comments can make the difference between code that is easily understood and code that remains a mystery, potentially leading to errors or unnecessary rework. However, it is crucial to recognize that an overreliance on comments may indicate that the code itself is too complicated[11]. Whenever a developer thinks to write a comment explaining their code, this might suggest the code they have written is too hard to understand. This might mean that their code will suffer under maintainability and readability. Ideally developers wouldn't need to write any comments as the code in

itself it self-explanatory, but we don't live in that world unfortunately. This lab made me think twice about commenting and I will now catch myself if I feel the need to add one and read over what I have written to maybe refactor my code so it's easy to read and understand.

Another code smell that I learned about in this practical is feature envy. It occurs when a method in one class excessively interacts with the data or methods of another class, reflecting a misplaced responsibility[9]. This behaviour can be thought-provoking, as it challenges our conventional understanding of object-oriented programming, where the fundamental principle is to encapsulate behaviour within the appropriate class[12]. When a method exhibits feature envy we have to ask ourselves if the responsibility of the classes is distributed as it should. This issue emphasises the importance of evaluating class design and allocation of responsibilities in order to keep the code clean and maintainable. This code smell, helps keep developers up to standard on the principle of object-oriented design and overall quality of the code. It can be hard sometimes to catch this code smell, as logically a method belongs in a certain class, but as we implement it "takes" from another. To not let this happen, we need to keep checking over the code we have written that it follows all the software design principles. During this practical, I realised just how important code smells are, they won't break your code, but are usually the cause of it way down the line after multiple changes around the codebase. I will be vigilant in catching myself, to not make such mistakes, and if I do, I know what to do….refactor!

During the second part of the refactoring practical I refactored the existing codes by completing all the questions from question 7 to 11. The main part of the refactoring was applying the strategy pattern. We used this, as price and travel points are frequently changed in the code and this makes it a way more seamless and easier process. This enabled the selection of the algorithm at runtime, promoting loose coupling and extensibility[13]. As we upgraded this code we changed from an entangled and strongly coupled code to a maintainable program. Encapsulating each algorithm as a separate class, is a lot better for maintainability as we only need to modify this class for changed functionality, or we can add an entirely different strategy. This behaviour follows the principles of Single Responsibility Principle (SRP) and the Open/Closed Principle. This enables us to change between a multitude of strategies used by our program at runtime. This flexibility is advantages as the choice of which algorithm we will use is only known at runtime or when the user indicates which one the program should use. This dynamic behaviour fosters adaptability, as highlighted by Freeman and Freeman in their book[14]. This way of developing the code also leads to another major advantage, testability. We can easily test each strategy independently, to easily find error in the code. This promotes the use of Test-Driven Development. Reusability is obviously a major improvement as we implement the main functionality in the one place, rather than all over our project. However, this approach also leads to some limitations and challenges. One general complexity is the greater overhead, but this is mitigated in this case, as the program is relatively small and simple to understand and maintain. This could be more cumbersome with larger projects, but I feel like this negative is vastly overshadowed by the improvements in maintainability, reusability and testability. Another general limitation that I see with this approach is that it may not be the perfect solution for all applications. When this behaviour is used in small and simple programs, this may add unnecessary complications. In our use case as we are learning to use these techniques, this use is more than justified, but if we were developing this program for personal use, we may not need to, or it may be a lot simpler to just add the functionality of the code inside the program files.

It's been very enlightening seeing how applying this Strategy pattern during this practical, as I saw how easily I can upgrade my applications to be more maintainability and easier to use. Moving forward I will try to use this in my coding. These practical showed me how important is to learn these software principles and more importantly use them in your work. It's a crucial toolkit to have as a software engineer and I'm grateful I could have learned so much, as I feel like this has made me a more experienced and a better developer. I will go forward to industry and apply all these new skills I have learned in these practicals.

7 ChatGPT

7.1 Work Done

In this practical I used the chatgpt model to help me program a game. We got the requirements of the game at the start of the practical and I created prompts for the AI to complete, so bit by bit we created the entire working program. The I used the AI model to extend the functionality of the program we just created, by querying the model multiple times, so we can get the program to work without any mistakes and major logical oversights.

7.2 Reflections

When developing the prompt for the AI, I firstly created prompts that are more conversational and sounded more like I was describing the problem to a human. I found out pretty quickly that this method will not work as the output was less than desirable, as it didn't quite understand what the question I was asking is. During this practical I was using the gpt-model 3, which performs a lot better by providing it clear and concise prompts[15]. Creating a good prompt was challenging because it required me to think critically about the problem and ensure that I provided enough information for the AI to understand my needs without overwhelming it with unnecessary details. I had to consider the AI's knowledge limitations and potential misconceptions when creating my question. The criteria I had for a good prompt are clarity, completeness, relevance and structure. The prompt needed to be easy to understand and not include any ambiguity as this caused wrong results. The prompt needed to have all the crucial information for the AI to create the program, like the classes and the language, which in this case was Java. The question focused on the core problem and was focused on the Object Oriented part of the program. Finally I structured my question so it was as easy to follow as possible and not include any misleading or confusing parts, where the AI could trip up on. Later I used the gpt4 model, which was quite a difference[16]. I didn't follow my criteria this time and had a more conversational prompt and the model performed a lot better. The application was built completely after two prompts, which was outstanding to see. That it could easily and correctly understand a longer and conversational prompt and turn that into the exact working code. The second prompt was to correct some minor issues like the frog hopping out of bound and getting the dimensions from user input, but nearly the entire program was made with a single command.

Using the gpt3 it took a couple of prompts to get it working. It didn't take a lot of time at all, most of the time was waiting for the response to finish curating. All of the code written by each prompt was correct, but I needed to be very specific, hence the greater amount of prompts I needed to use. I needed the AI to iron out some problems it didn't see firstly when writing the classes, like the frog being out of the grid, or the input not being valid. However when I used the gpt4 model, the program was practically completed in an instance, I added on more prompt after the first one to iron out the same problems as I stated before with the previous model. The time to create this program was greatly shortened by a huge factor, even with the gpt3 model, I didn't need to use my brain at all. I didn't think of any logic, I didn't consider the edge cases, I didn't think long and hard about the problem, I just asked the ai question and it answered with code, that was correct and needed to be added to the program. The huge increase in time savings that the gpt4 model caused, makes me wonder how the next model will improve, which is exciting to think about. Developers are going to be more and mor productive, by saving more and more time on "simple" programming tasks[17].

Extending the program was where this model didn't perform as well. The new function it created had many oversights and it used the old die with only 4 options. I needed to keep reminding it of the mistakes or oversights it was making. This time the model felt like I was holding its hand as it kept making small mistakes. This may be because the Ai couldn't look that far back and have a full clear idea of the entire application to make big structural changes to the app. These models are not great for structural changes as they are not great at seeing how the full application behaves with multiple classes and I'm sure in bigger systems it might

not even be that useful. This is the main flaw of these AI models. In huge applications with thousands of lines of code, it would only be helpful to use when extending single classes by some simple functions, but I don't think it would be very useful if we asked it to change multiple classes that all interacts with each other in a complex web of interactions. For simple changes that don't change the structure of the project the AI produces really good results, but right it's not yet ready to tackle changes to huge applications.

I agree with Kents's quote, as it did help a lot with programming this program, especially the gpt4 model, but its limited to only an assistant. Right now it's great at answering basic programming questions and creating some basic to mid complex solutions. It does need to be overlooked a lot as it has a tendency to produce wrong code that simply won't work or has many glaringly obvious problems. However these models at the hand of a person with no programming experience are basically useless as they won't know how to look for these oversights and problems. They don't how to even run these, how this code works or even they won't know the correct questions to ask. Most of our skills are easily replaceable by these model, like simple programming that involves some critical thinking, but the idea of specialised programming and understanding of programming concepts have raised in value. An experienced programmer is need to overlook these systems to make these systems effective tools that greatly increase our productivity. This was proved during this practical as the answers to my question had many issues that I needed to spot and needed to know the right questions to ask to solve them. Right now I think that this AI is not our enemy that will replace us, but as a tool that will greatly increase our productivity, by taking out the mundane tasks and simple programming that we had to write ourselves. Now we can focus on the complex issues that the AI can't yet handle.

References

[1] Michael Bloch, Sven Blumberg and Jürgen Laartz, Delivering large scale IT projects on time, on budget, and on value, McKinsey, October 2012, https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/delivering-large-scale-it-projects-on-time-on-budget-and-on-value

[2] Rachel Oliver, Why the software industry has a love-hate relationship with UML Diagrams, December 2022, https://creately.com/guides/advantages-and-disadvantages-of-uml/

[3] Aniche, M. F., & Gerosa, M. A. 2010. Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers. Third International Conference on Software Testing, Verification, and Validation Workshops , 469-478

[4] West, D., & Grant, T. 2010. Agile Development: Mainstream Adoption Has Changed Agility. Cambridge: Forrester.

[5] Janzen, D. S., & Saieden, H. 2007. A Leveled Examination of Test-Driven Development Acceptance. 29th International Conference on Software Engineering. IEEE

[6] George, B., & Williams, L. 2004. A structured experiment of test-driven development. Information & Software Technology , 46 (5):337-342, 2004

[7] Kaufmann, Reid and Janzen, David, Implications of Test-Driven Development: A Pilot Study, 2003, https://doi.org/10.1145/949344.949421

[8] SoapUI. 2019. "API Mocking: Best Practices & Tips for Getting Started." SoapUI, SmartBear.

[9] Fowler, M., 2018. *Refactoring*. Addison-Wesley Professional.

[10] McConnell, S. (2004) *Code complete: A practical handbook of software construction, Second edition*. New York City , New York: Cisco Press.

[11] Beck, K. and Andres, C. (2008) *Extreme programming explained: Embrace change*. Boston, NY: Addison-Wesley Professional.

[12] Gamma, E. (1995) *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

[13] Gamma, E. *et al.* (1993) “Design patterns: Abstraction and reuse of object-oriented design,” *ECOOP’ 93 — Object-Oriented Programming*, pp. 406–431. Available at: https://doi.org/10.1007/3-540-47910-4_21.

[14] Freeman, E. and Robson, E. (2004) *Head first design patterns: A brain-friendly guide*. Sebastopol, CA: O'Reilly, Edition: 10th Anniversary ed.

[15] Wang, S. and Jin, P. (2023) “A brief summary of prompting in using GPT Models.” Available at: https://doi.org/10.32388/imzi2q.

[16] Heaven, W.D. (2023) *GPT-4 is bigger and better than chatgpt-but Openai won't say why*, *MIT Technology Review*. MIT Technology Review. Available at: https://www.technologyreview.com/2023/03/14/1069823/gpt-4-is-bigger-and-better-chatgpt-openai/

[17] Jee, K. (2023) *How I save over 5 hours every week using CHATGPT as a data scientist*, *Medium*. Level Up Coding. Available at: https://levelup.gitconnected.com/how-i-save-over-5-hours-every-week-using-chatgpt-as-a-data-scientist-715fb5fd68d