COMP47480 Contemporary Software Development

Lab Journal

Sukrat Kashyap (14200092)

BSc. (Hons.) in Computer Science

UCD School of Computer Science
University College Dublin

April 27, 2018

Table of Contents

1 Extreme Programming 2

1.1 Work Done 2

1.2 Reflections 2

2 Modelling with UML 5

2.1 Work Done 5

2.2 Reflections 5

3 Test-Driven Development 7

3.1 Work Done 7

3.2 Reflections 7

4 Object-Oriented Principles 9

4.1 Work Done 9

4.2 Reflections 9

5 Refactoring 12

5.1 Work Done 12

5.2 Reflections 12

Practical 1: **Extreme Programming**

In this practical, we role played and tried to understand the iterative development also known as agile development in Extreme programming.

1.1 Work Done

We were a group of 5 people. Our task was to build a fridge using the Extreme programming methodology. So, at the start, we divided our group into 2 consumers and 3 developers. I was one of the developers in the Iteration 1. Consumers were then supposed to create stories telling about the features they want in the fridge. These stories by the consumers were given to us to give the expected time to finish the feature. We developers then estimated the time required for the stories and gave the cards back to the consumers. Some stories were not clear like "It has two-sided door". So, we asked the consumers whether they want the doors for both the compartments or one big two-sided door for both the compartments. After clarifying this detail, we drew a small picture of the same to make it clear. After giving the cards back we gave our work time of 3 minutes for the iteration. The cards were given back to us and us three developers one by one took stories and implemented them.

In similar fashion, we did the second iteration. Only this time the roles were switched. So, then we wrote the features on top of the fridge that was already being created. The developers clarified some details such as how big TV they want. The estimates of the stories and their total work effort were given. We then picked all the stories as the estimated time of the stories were completely fitting their effort time. They then took the stories one by one and build the device on the previously implemented fridge.

After, the 2 iterations we retrospect on the whole process. The customers were happy with the product. Developers had few difficulties implementing as some of the requests were bizarre but doable. Queries were solved satisfactorily but most of the decision was given to developers as what they might seem fit.

1.2 Reflections

Extreme Programming is one of many software development methodologies which aims to improve software quality and responsiveness to changing customer requirements. It is a type of agile software development which has both incremental and iterative properties. These properties tackle the varying project requirements (One of the major problems in software development).

In the practical, we used Extreme programming methodology to develop our fridge. Only 2 iterations were performed which was enough to give us an insight into the development process. The difference between two properties namely *Incremental* and *Iterative* was also clear. In simple terms, *Incremental* was adding new features to our fridge and *Iterative* was adding these new

features repetitively. It also helped us to understand the developers and consumers mindset. I felt that it helped me understand the 4 main agile manifestos in practice:

Individuals and interaction over process and tools EP felt like a program which decoupled two entities (business and development) in software development and created a link between the two.

Working software over comprehensive documentation While developing fridge there was no guideline of documenting or even thinking and caring about future needs but it strongly withheld the idea of testing each feature before saying its complete.

Customer collaboration over contract negotiation The developers collaborated with the consumer to understand the feature of the fridge before estimating their time and communicating when implementing the same.

Responding to change over following a plan Few modifications of the original feature was done to accommodate new feature in the second iteration.

One of the other advantages that I felt was that the process was much simpler to understand and implement. Unlike other Agile practices such as Scrum. Scrum seems a bit complicated as it contains various roles such as Product owner, scrum master, team members etc. and contains multiple artefacts such as product backlog, sprint backlog, burn down charts, Taskboard etc. Whereas, when it comes to EP the process is much simpler having only 2 roles that are the developers and consumers with the artefacts being the story cards. In all my internships scrum was the main methodology of development. It seems to have fine detail about each role and seems to be a bit less flexible. Whereas the EP abstracts the management of each role. Giving the flexibility to the developers and consumers to manage their internal affairs on their own. EP seems to look like a collaboration and interaction process between the developers and consumers. While saying this, EP does have guidelines which the 2 entities must follow for better throughput [1]. For e.g.

- For developers
  - Pair programming
  - Testing first approach also known as (TDD Test driven development)
- For consumers
  - Stories must be a feature that the consumers with no implementation detail
  - Stories chosen to be developed must not be more than the promised effort number given by the developers

The incremental property of Agile practices, in general, gives the business side of the company to evaluate their product in the market after every release which helps to reduce the cost. Unlike the Waterfall process where the defect is usually found in later stages, making the process revisit analysis, design and implementation part. Whereas in an agile process, the defects are found early, giving the opportunity to the business side to re-evaluate their product. Which usually involves in a spin-off of the already released product and adding new features to counter-attack the problems. E.g. we had an ice and iced water dispenser in the fridge and if it didn't seem to work to attract customers, in iteration 2 we added ice cream maker to attract the customers.

In the end, it felt Extreme programming was a nice way to development. But still, it lacked stronger and finer guidelines and rules when it came to a bigger team. We were a group of 5. So, it seemed easy to follow and implement the ideology. But it seems to be naive when it comes to

bigger teams on the much bigger project [3]. Since its iterative, it seems harder to create software which has deadlines. One of the problems that I realize in Agile, in general, is that when the developers start working on an iteration the stories are locked and not allowed by the consumers to change. So, a major defect or bug in the system which needs immediate care during the iteration is harder to include. Also, most of bugs or defect are minor and doesn't need many changes but some of them are major and it becomes harder to estimate the time to fix. The most problematic part of the EP is the estimation of stories, in EP the developers estimation is somewhat vague and is not always true. EP says to finish the story for the time being estimated. But, it doesn't happen. Estimated time is mostly underestimations and it is due to different developers have different speed in development. Even though other agile methodologies sometimes use a vague numbering system for estimation called story points. If the stories are not finished then they are moved to the next iteration/sprint. A team's number of average story points achieved in a sprint in the past and their current estimation is used as a measure to estimate the total work effort of the team. This seems a better approach than asking the developers their effort time for an iteration. Overall, EP is a more engineering focused methodology which seems to work best on small teams.

Practical 2: **Modelling with UML**

This practical we learn modelling the software using use-case model, domain model and interaction model.

2.1 Work Done

In this practical, we were given to create Use-case model, Domain model and Interaction model of a library system. The library had books and journals which could be borrowed by the members of the library (Students and Staff). There was a restriction on the number of items each member could borrow. Journals could only be borrowed by the Staff members. There were two types of loan for the books namely short-term (4 hours) and long-term (4 weeks). We had to represent the system in 3 different model representation. We first created use-case model for the library system which was done by finding different actors (student, staff, and member) and use-cases such as borrow journal, check availability and common use-cases such as borrow book, return, and display. After the use-case model, we picked out nouns suitable for making classes like Student, Member, Staff, Copy, Book, Journal and created links between them. We then created an interaction diagram which showed us scenarios of communication between the classes and had to add a new booking system to our domain model. We then were told to add a new requirement of notifying the members if they had the book or journal for too long. To which we added a booking system which notified the member when the booked in all of our models.

2.2 Reflections

We started modelling our library system by first created use-case model based on the description. For which we had to find out the actors and use-cases. We thought of actors as the staff and students. Our use cases were borrowed, return, check availability and display. Then we tried to reduce the functionalities by excluding and include generalization. Student and staff were generalized using the actor member. This reduction of both the actors to members was because they shared mostly all the common functionalities except borrowing journals. All the use cases were included in the display because they shared functionality. An actor performs display then borrows/return and then displays new changes in the system. Borrow journal use-case was given separately to staff as only they are allowed to borrow journal. A common return was used instead of return book and return journal as by using simple logic as to when a student cannot borrow journal, it won't be able to return it either. Hence common return for both made sense as they can only return what they borrow. This form of representation seemed quite useful as it was able to showcase various parts of our system in more readable format. A paragraph of description is hard to understand in one go. This form of modelling enabled to look at the system in much user-friendlier format. Regardless, it failed to showcase much tinier important detail such as students are only allowed to have 3 books and the types of loans of the book available.

The second form of modelling which was done using the description and the use case model was

a more of a class diagram that would have been implemented when actually writing the code. It was done by first identifying the nouns. In our case, we thought of Student, Staff, Member, Book, Copy, and journal as our nouns. The second step involved finding the relationship between the same. Student and Staff inherited from the members. This was thought as library system should just care about the member and the limit imposed on them should be handled by their concrete classes student and staff. The reason Copy was created as the book could have multiple copies with a different type of loan but all the other detail would remain the same. Hence N:1 relationship was identified in the Copy vs Book. Since journal didn't have any copies so the copied class was omitted from the journal. The borrow and relationship were straightforward from our use-case model. Comparing the two models above, we see use-case model tends to just identify the actors and what they did irrespective of how they did. Whereas, domain model representation adds a technical aspect of the system. This could be easily seen from the attributes corresponding to each class. Both representations are quite similar to domain model being a bit more technical. Domain model could also show them the fine detail of the limit on each student and staff by assigning a constant value of 3 and 12 respectively to the limit attribute they both shared. We decided to skip such fine details. But it gives us insight into the level of detailing without making the model too complex can be achieved. Although, it seems like a better option than domain model. I feel Domain model has the potential to become too complex real quickly than use-case model.

The third form of modelling was Interaction diagram which tries to picture the exact interaction between the general ideas but more in sequential form. We took member, booking system and item to show the interaction. It shows the sequence of getting items from booking system which in turn called each item and their info. It returned a list of items and their status. Borrow item can be issued later passing the exact item one wants, the result was the confirmation. Notify was included as this was the third requirement which would be done timely until the user returns the exact item. Our sequence diagram was not as perfect as it should be. But I believe it did represent the sequence in the steps for e.g. return can only be done if borrow for that item is issued and notification will be done until the item is returned. Notification step was also added to other diagrams as an addition which didn't seem to be much of a hassle.

After building all the three models, one can see that the use-case model represented the system in a vague manner missing all kinds of detail and giving a very high-level overview. The domain model gave simple and fine details and also somewhat the implementation of the same whereas sequence diagram gave those implementations and relation in the domain model a sequence in which to be performed. But when it comes to usage of these above three models in real life, I believe it is not as useful. I believe these models were highly required and recommended in the past because of there were none to very few development tools and required a great deal of work to manage and learn about the new system which was already implemented. In today's world, we have so many advanced development tools such as IDE, debugger etc. that it has very much diminished the use of modelling. Also, the changes in the project are very rapid making models before coding redundant. A great example to show this case would be finding the implementations of an abstract class. Before IDE's finding a class in a big system which implements certain class was extraordinarily time-consuming. Hence, the models such as UML helped to figure them out faster. But in today's computing finding the implementation is a task of a mere single click. With that being said, simple to represent the whole system still exists in form of vague pictorial drawing rather than following a standard.

Practical 3: **Test-Driven Development**

This practical involved using JUnit with Eclipse. We used these tools to perform and understand Test-driven development. This technique was done on small programming tasks.

3.1 Work Done

In this practical, we had to use the TDD (Test driven development) approach to develop a piece of software. The first part required us to create a class which returns true for a temperature less than or equal to 0.0 and false otherwise. Since we had to use TDD approach we first created a class to test that the class function returns true. The test failed because we had no implementation the first time around so we implemented just enough code to pass the test on the second try. Then we wrote another test to see if the code returns false if the temperature is greater than zero. Since the test failed again, we wrote more production code to pass the test which completed our first part. The second part involved creating another software piece in which we pass the length of sides of the triangle and then check the type of the triangle with an isValid function. In the artefact pdf within this folder, one can see the step by step iteration of the TDD process that was employed. This basically involves writing a test which tests one part of the requirement and then writing production code to pass the test that previously failed. Code coverage was run to check how much of our test covered the production code. Since it left out a couple of branches in the "if else" condition, we created more tests to cover those conditions.

3.2 Reflections

Testing is a very important part of the development process. Many ways have been proposed to write and maintain the test. There are even many different types of testing for e.g. Black box testing, white box testing, integration testing etc. All of the types try to achieve the following objectives [4]:

- To give the programmer the confidence in his code and hence removing the scary part when adding or updating a system.
- To keep in check of the bugs that have been found earlier.
- It does take more time in developing than only writing production code but it saves a lot of unnecessary need of debugging and finding failures later on.
- To ensure many properties of software like correctness, reliability, performance etc.

In this practical, we tried out one of the methodologies of testing known as TDD (Test driven development). Unlike traditional development, where we write code then write some test against it. In TDD, we write test according to the specification and then we write the code just to pass

those test. It is followed by refactoring to clean up the code. In the practical, we followed this pattern to iteratively develop the Weather class in Part 1 and then we built the Triangle class according to the specification in Part 2. Each test case when written followed or tested a part of the specification. This part of the system was built in order to pass those test. No code more than sufficient to pass the failing test was written. Then in the next iteration, another test case was coded up which tested a different use case of the specification, followed by writing production code which passes the currently failing test without failing the previous ones. One can see the iteration in the artefact.pdf within this folder for the practical. Code coverage analysis was run later to check the level of code coverage by my test cases. It was a little below 100% as we left some branches in the 'if' condition for scalene and isosceles triangle due to the short-circuiting and combinatorial nature of boolean values. It seemed like maybe the TDD approach was not done properly. But in fact, the TDD was used correctly. It was just nature of the program, that required testing the same function by passing parameters in different order to pass through the short-circuiting and combinatorial nature of boolean expression. This was solved with the writing some additional tests.

To understand better about testing and assessing the test case (i.e each test case contribution to the coverage and checking a functionality). We tried to find a redundant test case by removing each case one at a time and checking the code coverage. The code coverage in my case did change whenever any test case was removed. Proving the contribution of the test case. We also tried to validate the test case by changing the method slightly from the specs which resulted in failing test assuring the contribution of the test case.

In my experience, I have created many tests against programs that are already developed or programs that are being developed. I have used TDD before but not to the extreme extent. It is known that test gives us a confidence in the program created. But what about tests, Tests are written code as well. How can we know that the tests are working fine? For this, we use mutation testing. I have done mutation testing before. But I never knew the name of the same. It is a more intuitive way to understand the test, I believe. In the practical, we performed mutation testing by changing a piece of code and running the test. If they fail, then it means the tests are working. I feel most of the programmers do this intuitively by changing the code and testing the tests. But the practical made me understand in more depth how to perform and analyse.

Overall, TDD seemed a bit tedious approach. But it proved to have a number of qualities. Because of TDD, we did not write a single line of unnecessary code which we usually do when designing and writing the code first. It also did not let us wander away from the specification into writing code thinking too much about the future. Since we were writing test's first it helped us to write code such that it was easily testable. The only downside that I felt was that the languages that we currently are using like JAVA do not help us with our TDD development because of its strict type in nature. Since we couldn't even compile our first test. Dynamic programming seems to support this TDD approach lot better. But at the same time, it makes the program too error-prone. Hence, TDD is an amazing approach but I feel we need a language that supports this methodology.

Practical 4: **Object-Oriented Principles**

This practical involved understanding and applying object-oriented principles. These principles are applied mainly to Object-oriented programming. Java was used in the practical. But language is immaterial to the principles.

4.1 Work Done

The first task was to find the violation of the open-closed principle of PostageStamp and Square Stamp such that PostageStamp is closed for client change which is the Square and open for extensions. The violation found in the OCP.java was that the PostageStamp was tightly coupled with the Square class. This tight coupling closed the PostageStamp class for further extensions as changing the shape of stamp requires changing the PostageStamp class itself. To fix the issue, an interface named Shape is created which becomes the argument and shape of stamp for the PostageStamp. Now, other classes which can act as a shape to the PostageStamp class must implement the shape interface and can be passed without changing the PostageStamp and is extensible.

The second task requires finding the violation of Single responsibility principle. We know that the Hexapod is actually a dog and a human. Hence, we created two classes namely Dog and Human which dealt with task related to themselves only. These two objects were then created and combined by the Hexapod. Hexapod could be removed from the project itself. But was chosen to keep because the requirement doesn't want us to fix the code. The principles are what we were looking at. Since Hexapod could be thought of as a requirement. We decided to keep the Hexapod and just put the responsibilities to their respective class and use it in the Hexapod.

The third task, wanted us to find and correct the violation of the law of Demeter. We found that Customer was depended on Wallet. The Shopkeeper was depended on the Customer. But the Shopkeeper was accessing the Wallet class on which it was not directly dependent and hence violating the law of Demeter. To fix the issue, 2 new methods corresponding to the usage in Shopkeeper class were created in Customer and used in Shopkeeper class making dependency of Shopkeeper only up till Customer. The Shopkeeper was also checking whether the customer has the money or not. But by the single responsibility principle shopkeeper should not care about checking the money, it should rather just ask for the total money.

4.2 Reflections

The co-author of the Agile Manifesto Robert Cecil Martin is the promoter of SOLID design principles [2]. SOLID is mnemonic for five design principles for object-oriented programming. It is intended to make software designs more understandable, flexible and maintainable.

The five design principles are as follows:

- Single responsibility principle: A class should have only a single responsibility. The responsibility should be entirely encapsulated by the class.
- Open-closed principle: Software entities must be open for extension but closed for modification.
- Liskov substitution principle: Objects in a program should be replaceable with the instances of their subtypes without altering the correctness of that program.
- Interface segregation principle: client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle: objects and specification should depend on abstractions and not on the concretions.

There were 2 other principles that were shown in the class. They were "No concrete superclasses" and "Law of Demeter". No concrete superclasses recommend that all superclasses in a system are abstract and Law of Demeter states that each unit should have only limited knowledge about other units that are only units closely related to the current unit. It also states that each unit should only talk to its friends and not to strangers and can talk to their immediate friends.

In this, practical we worked on three design principles out of the 7 principles stated above namely: Single responsibility principle, Open-closed principle and law of Demeter.

First, we worked on Open-closed principle. I felt Open-closed principle extensively uses the concept of polymorphism. Polymorphism is an object-oriented ability to process differently depending on their data type or class. Open-closed principle uses this nature of OOPs to allow open part of the principle. The open nature refers to the open to extensions. Using polymorphism we are able to extend our code by creating abstract class or interface which then can be derived from many different concrete classes doing different things but coming under one name. Now to use this extension we need a module that uses the abstract class instead of the concrete class. This makes the module closed which refers to change in implementation should not affect the module using it. According to me, the open-closed principle is the most widely used principle as it is easy to implement. The only part that needs thinking is which class requires an extension or which class can have many forms and which class will remain the same and use only certain parts of the form. This principle is usually used closely with IoC (inversion of control) where the objects are created by an outside entity. All the concrete classes implement certain forms which are configured in the IoC container. Making the whole application closed whereas open for the extension at the same time as another concrete class can be created and replaced with the old concrete class by changing the configuration of the container.

Second, we worked on Single responsibility principle. One of the hardest principle to bring into practice, according to me. This principle uses the encapsulation nature of OOPs by stating that each module or class should have responsibility for a single functionality provided by the software. This principle is best applied in places where a number of small modules are required which have almost no connection or dependency between each other. The hardest task of this principle is to isolate the different mutually exclusive responsibilities because the software is mostly an intertwined mesh. Single responsibility principle is applied to interlinking modules as well the only condition is that the services provided should be narrowly aligned with the responsibility. The problem, in this case, is to find those isolated narrowly aligned responsibilities. Dividing responsibility can sometimes lead to too many classes and modules which later becomes overhead to manage. Even though this division actually makes the software more robust and makes the testing easier.

Third, we worked on Law of Demeter, this law constricts the usage of non-close objects. It basically restricts unnecessary dependency that usually a software easily has due to easy access of those

unnecessary dependencies from real dependencies. I feel it helps in making more maintainable and adaptable software as due to this restriction unnecessary dependencies are blocked. Due to this blocking, changes made in one object require changing to code to only classes which directly depends on it. Otherwise, one small change in one class would lead to changes in several classes that the module does not even depend on properly. I feel even though it does welcome changes, but it increases the number of wrapper methods to propagate calls to non-dependent components which leads to wider interfaces.

In the end, every principle seems to have some advantages and disadvantages but I feel they are a good way to start any project as in the start projects are ever changing and too iterative. And these principles protect a lot from many of the unseen problems in the future. Since these designs have some overhead and in future can obstruct the development process. Refactoring and better design according to the system need is a must where one has to deviate from the hard fast principles for better development.

Practical 5: **Refactoring**

The purpose of this practical was to experiment with refactoring. Some hints and reasons for refactoring was already given and then moving forward it was left on us.

5.1 Work Done

In this practical, first, we tried to find out the long method using the JDeodrant. In that, we found the statement function in the Customer class. We extracted out the switch method calculating the rent for the car from the rental object. This switching method was violating the law of Demeter by accessing Vehicles field. Then we ran the feature envy of the JDeodrant, which gave us the recommendation to move the newly extracted method to the Rental class as it used mostly the Rental class and did not access anything with the Customer class. After this, we realised that the law of Demeter was not resolved for some part as there was still frequentRentalPoints calculation that was breaking the law. We moved this as well to the Rental class by following the extract and move method of the Eclipse IDE. This resolved the problems from the Customer class. We created two private methods for calculation of the totalRental and totalFrequentRentalPoints just to make the code clearer. Switch case was still a code smell in rental class. So, we used strategy pattern and removed the comparison of the vehicle model type by implementing concrete classes for each type of Vehicle, making Vehicles an abstract class. Then logic of calculation was in each of the concrete classes with Rental not caring about its implementation. We also implemented the htmlStatement method returning the statement in HTML format.

5.2 Reflections

A lot of refactoring was done in the practical for the code provided. Code refactoring is the process of structuring the code such that it is easier to understand, reduce its complexity, improve source code maintainability and improve extensibility. It is usually motivated by noticing code smell or when it violated or does not follow principles like SOLID for OOPs etc. In this practical, we actually solved a number of issues like long complex method, feature envy, the law of Demeter etc. The important part of refactoring is to improve design and structure without changing the behaviour. To help us give the confidence that our refactoring does not break the functionality. We already had tests written against the main class. The refactoring process is also important. I feel refactoring should be done in stages and not in one go. Stages make sure you are not breaking any previous code. It also helps in reverting back as sometimes refactoring can increase the complexity which we realise after some time spent in refactoring the code. The reasons for this mostly is because of exceptions in the business rule which makes it harder to find a uniform design to tackle the problem.

The first task was to break down the complexity of the statement function of the customer. The function was too long accessing many fields which it does not even relate to (Law of Demeter). The approach I took here was to first divide the function into many small functions. First for

calculating the rentalAmount and second for the frequentRentalPoints. After doing so we realized that the newly created methods actually do not interact with the Customer object at all and only relies on the Rental object. So we moved it to the rental class. This simple approach actually led us to remove violation of Law of Demeter as well as it reduced the complexity of the statement function. Most of the refactoring is like a chain reaction. One refactoring leads another and this way it helps cleaning up the whole code. Hence, it also makes sense to do them in stages. This refactoring was the most important refactoring for further extensions on Customer class. Since the public APIs did not change at all the test could be run in order to check for breaking changes.

Next stage of refactoring was in the rental class. The reason for this was, there is a high probability of changes in the way the points and the amount were calculated on Vehicles. Keeping all the complexity in one place leads to bugs and decreases code extensibility exponentially. Every time change in the calculation will be either the formula or addition of another vehicle. This would involve adding extra cases for switch statement and adding new fields for the model in vehicle class. The refactoring here is usually domain driven meaning it depends on the business model of the company. As the calculation is a business logic rather than coding logic. The rate of change in business logic is always more than the design, tools or the algorithm of the software. To help with this situation I thought and made an assumption that the rental is based on vehicle and days. So when rental wants to calculate the price it should actually ask the vehicle itself the cost. Now since cost has a dependency on the number of days, it is trivial matter as the rental could easily ask the vehicle the cost by giving the object the number of days. Hence, a concrete class was created for each vehicle type (getting rid of the type of vehicle as the rental does not care about type and only cares about the cost). Each class contained their own calculation of the amount for the number of days which was passed in by the Rental class. This design is also commonly known as strategy pattern.

The strategy pattern for calculating frequentRentalPoints. This strategy pattern, of course, has its own advantages and disadvantages. It was preferred in this particular situation as all cars depended on days for calculation of cost and points. If they did not depend on exact same parameters then strategy pattern might have failed. I feel assumptions are the basis on which a pattern or particular design is chosen. Here our assumption was that the car rent only depends on the days. But if it seems to change in the future refactoring would be needed again. Hence, refactoring would never create the perfect solution rather than will create a maintainable software for a certain period of time and then would require refactoring yet again.

In the past internships, refactoring was usually given to interns. Being an intern, we were told about the refactoring to be done but we never knew what was the reason for doing so. Upon asking some reasons were given which did not make sense. But after this practical, the reasons became clearer and importance of refactoring was known.

Bibliography

[1] Lee Copeland. Extreme programming, Dec 2001.

[2] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[3] Matthias M Müller and Walter F Tichy. Case study: extreme programming in a university environment. In *Proceedings of the 23rd international conference on Software engineering*, pages 537–544. IEEE Computer Society, 2001.

[4] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.