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 . . . 3

1.1 Work Done . . . 3

1.2 Reflections . . . 3

2 Modelling with UML . . . 6

2.1 Work Done . . . 6

2.2 Reflections . . . 6

3 Test-Driven Development . . . 8

3.1 Work Done . . . 8

3.2 Reflections . . . 8

4 Object-Oriented Principles . . . 10

4.1 Work Done . . . 10

4.2 Reflections . . . 10

5 Refactoring . . . 13

5.1 Work Done . . . 13

5.2 Reflections . . . 13

6 Design Patterns . . . 15

6.1 Work Done . . . 15

6.2 Reflections . . . 15

7 Company Seminar Series . . . 17

7.1 IBM seminar . . . 17

7.2 Speaker . . . 17

7.3 More than Software . . . 17

7.4 More than Engineering . . . 17

7.5 Lifecycle of Project . . . 18

7.6 Manage Evolution . . . 19

7.7 Q & A . . . 19

7.8 Facebook seminar 20
7.9 Speaker 20
7.10 Report 20
7.11 Q & A 22
7.12 FoodCloud seminar 23
7.13 Speaker 23
7.14 Report 23
7.15 Comparison of Company Seminars 24

Practical 1: **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. Scrum 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. 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.

At 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. 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**

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 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**

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:

- 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.

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**

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.

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.

4.2 Reflections

The co-author of the Agile Manifesto Robert Cecil Martin is the promoter of SOLID design principles. 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**

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.

Practical 6: **Design Patterns**

6.1 Work Done

In this practical, our task was to use observer pattern to solve a particular problem. The Person and the AlarmClock were acting as data classes and all the work was done in the AlarmApplication making it have god class tendencies. To solve this problem, we created an abstract class for the Subject which acts the source of broadcasting when something happens. The broadcasting is done to all the interface Observers which have subscribed to listen to the Subject. These 2 were extended and implemented by AlarmClock and Person class respectively. AlarmClock calls the Observer update method upon a change in its time. Person class on update checks whether the AlarmClock has reached the alarm time if it has it wakes itself up. The AlarmApplication was then made to only instantiate the objects, register the person and call the tick method. Since we had to notify the person whenever the alarm time is set as well. The Person class this time had to store the wake-up time as well. For this reason, we changed the update method to pass in the object for pulling the state and an args object for pushing the event name. This was done so that the Person knows which event is being called and act accordingly. Cron class was extended on Observer which also listened to the Clock and acted when the time was right.

6.2 Reflections

Design patterns are a general reusable solution to a commonly occurring problem within a given context. It is not the absolute truth in software development but a template. In this practical, we tried to employ and solve the problem using the Observer pattern. Observer pattern is a behavioural design pattern that deals with common object communication. We were also dealing with an object communication between the alarm clock and the person to get up when the alarm is reached. Observer pattern uses an object called subject which maintains the list of observers who have subscribed to the subject and notifies the observers upon a change in the state of the subject. This notification is done via calling one of the methods of the observer. I feel observer pattern is great to solve a number of simple communication problems but also has a number of downfalls.

In the practical, our subject was the alarm clock and the observers were the person and cron objects. The person and cron were subscribed to the subject by adding them to the observer's list held by the subject. Our state change in the case of the alarm clock was the tick. The complexity increased when the person class also wanted to know whenever the alarm time is set. Push and pull are 2 different ways in which the observer can know about the state of the subject. To make things simpler and allow for both the methodologies. I made the update method to receive the subject and an args object to be passed. The pulling of the state is done using the subject-object received whereas the pushing the state could be done through the args object. This allowed the flexibility in the implementation of the subject in the alarm clock. Differentiating between 2 events coming from the same subject was another problem. For this, I made the person to pull the state from subject-object being passed and pushed the event name from the subject. This event name pushed was used as a distinguisher between the 2 objects.

When writing the code for the person class, I felt that the person could be made to subscribe any event due to this decoupling. Because of this, I added an if statement to check if the subject passed to the person is actually alarm clock or not. This was also done since we have a common interface for the Observer this forces us to cast to access alarm clock specific methods. The argument being pushed is an Object as well requiring another cast to string. This actually introduced coupling between the alarm clock and the person object but removed coupling from the notification mechanism. This seems quite troublesome but in fact, make sense as the person know that it should act only when a type of clock tells you the type rather than any other object.

Adding of Cron was trivial and was implemented without touch any of the other classes. The communication between the two objects could have introduced complexity and number of other issues in the program in the future. It was nice to see observer pattern solving the problem with allowing easy extension in the future. Personally, these design patterns are also useful in communicating the structure of the code to another developer. The problem which I think is hard to solve in observer pattern in automatic garbage collection is the holding of the reference in the subject class. This stops the collection of the observer when it's not needed anymore. The solutions to these are weak references but that is not a stable and correct method as the garbage collection is done by the VM at an arbitrary point in time. I believe a better approach is to unsubscribe them after use. In languages where garbage collection is not done, it's easier as a destructor can be implemented to remove its reference from the subject. The delete call on the object will first call the destructor removing references from the subject and then destroy the object itself.

In my experience, observer pattern has been used a lot where ever there is a requirement of acting upon a state change. This decoupling nature of the implementation of the observer from the subject state change allows for greater extensions. I also found that this notification mechanism actually saves quite a lot of computing as the observers do not have to keep polling for the change in the subject state as they are only notified when state changes in real-time.

Practical 7: **Company Seminar Series**

7.1 IBM seminar

7.2 Speaker

Paddy Fagan, Chief Architect at IBM Watson Health. He is working across SAAS and on premise solutions in healthcare and government. He has worked to bring together the customer team's requirements and the Product Development Organization to architect approaches and solutions that are the best fit for customers.

7.3 More than Software

He said that a successful piece of software doesn't mean a successful project. It was much more than only the software. It involved lot of different parts which had to work together to create successful projects. Some of those parts are as follows:

- Having the right offering: Giving something that the customers you are targeting actually need and will benefit
- Getting to the market
- Selling it: selling it at a price that the end users can afford it
- Operating it
- Supporting it: supporting your customers whenever fault occurs and also helping them to get used to eat
- Evolving it: while providing the services, the service should evolve according to the customer's needs

7.4 More than Engineering

Many other crucial professions/disciplines are involved in managing the Software

- Engineers: know what cannot be done
- Project Management: knows how to organise the project and how it should be worked towards

- Business: These people know what should be done
- Designers: knows how it should look like
- Test Operations
- Supporting: needed for helping the end customer with their queries
- Sales and Marketing: knows how to sell it
- Legal: department knows the laws and regulation that it should comply with

7.5 Lifecycle of Project

IBM uses Agile methodology in their lifecycle which involves Vision, Plan, Develop, Deliver and Operate.

7.5.1 Vision

IBM uses a Design thinking pattern which is process for innnovating and delivering fast. It adds certain practices namely hills, playbacks, sponsor users.

- Hills: it is expressed as an aspirational end state for users that is motivated by market understanding. It defines a mission and scope of a release. No more than 3 major release are recommended with a technical foundation.
- Playbacks: Moving forward requires a lot of feedbacks and that's where playbacks come into play. All design and deelopment work is iterative.
- Sponsor users: these are the people who are selected from real or intended user group. By working with sponsor users. It allows for better design experiences for real target users, rather than imagines needs.

7.5.2 Plan

Planing involves number of steps like mapping down story boards.

Their planning is similar to Agile planning where we have Epics (themes), Features (user identifiable features), Plan items (development iterations) and Stories (development sprints).

Stories are the leaf items that the Project team work on. Test, documentation and deployment is a part of every story and must be followed by the team members.

Planning is known as Playback 0. They use IBM Rational Team Concert (Jazz) tool to manage all aspects of their work, such as iteration, release planning, change management, defect tracking, source control, and build automation.

7.5.3 Develop

Development in IBM uses Eclipse as IDE, Ration Software architect (UML), RTC for source control, Tomcat as webserver, DB2 for database, JUnit for testing, Selenium for web browser testing, CheckStyle for static code analysis, Sonar Qube for continuous inspection of code quality like code smells and security vulnerabilities.

They use Jenkins/Build forge for continuous testing. Scripting is done using gradle and artifacts for managing software artifacts and metadata.

All the software is deployed for testing on WebSphere and DB2. Testing is an important part of development and a story or a feature is given clearance for deployment once when following checks are done.

- Functional Verification is equivalent program verification
- System Verification
- Business Verification
- Peer code reviews

This process is Playback N

7.5.4 Deliver

The projects follows continuous delivery and deployment. Releases are done every 2 X 2 weeks.

7.5.5 Operate

This involves multiple disciplines such as Deploy, Monitor, Support. Support organisation has multiple heirarchy e.g. (L1/L2/L3)

7.6 Manage Evolution

Release are done monthly Separate stream for parallel development Merging Streams Check points Legal clearance

7.7 Q & A

Some question that were asked at the end of seminar were:

1. Software project has 3 main characterstics namely Performance, Features and Stability. When and how one would prioritize them?

Answer: Priority depends on the stage and type of project. Usually in the start Feature is the most important to showcase in the market what new you are bringing, then it shifts to stability when more and more people start using it and at last when you have competitors as well the priority shifts to performance.

2. How to update/maintain the support team?
   Answer: Youtube videos, documentation or a group demo is done weeks before the release.

3. Pair programming?
   Answer: Pair programming is something that is up to the team. But usually teams donot prefer pair programming.

4. UML diagram in IBM?
   Answer: IBM uses tools to create the UML diagrams from the code like Rational Software architect.

5. Development methodology used in IBM?
   Answer: Very agile methodologies are used and Software like IBM Rational Team Concert.

7.8 Facebook seminar

7.9 Speaker

Richard Sheehan (Production Engineer) and Mike Elkin (Site Reliability Engineer) at Facebook.

7.10 Report

The speakers started with a statement which said that "Failure is not an option but it is inevitable". They meant to focus on the fact that Failure is a part of Software Development. It is something we definitely do not want but at the same time it is inevitable and cannot be deferred. The main idea behind was to accept this nature of Development and the team should be able to learn from them and make sure it never happens again.

They showcased the vast Facebook's data centre all over the world. Each datacenter was built in order to manage massive traffic coming from not only Facebook but from Instagram, Whatsapp etc. They also showed the switches that they build for the data centres which they chose over purchasing from the vendors because of the cost. Each data centre was supposed to have a massive infrastructure to handle large traffic from the network. Now with such big infrastructure possibility of error is vast. Identifying and resolving them is another big challenge. They gave us an example of how one of the errors that occurred shut down the entire Facebook data centre. Performance is also important and with such large infrastructure, it becomes harder to optimize and increase performance. They showcased their software development methodology by giving an example of improving their performance of data transfer between networks to users.

Their methodology does not include agile planning or software development. It consists of abstract project development method which is as follows:

1. Investigation
2. Analysis
3. Project Retrospect
    - Set clear project goals
    - If you cannot measure it, it did not happen
    - Get the right stakeholders
    - Requirements gathering
    - Career progression is not linear
    - Sunk cost fallacy (if not working move on)

The overall structure is quite similar to IBM's Vision, Plan and Develop. The words involved are quite and different and so is their idea behind it. IBM focuses more on software as building a machine whereas Facebook's development is in terms of numbers if a project tends to increase the performance they ask themselves by how much. Both have teams working on projects but whereas IBM follows their project lifecycle. Facebook leaves everything to the team. Facebook has more of an "ownership" way of development where they give the project to a team and then the project is owned by the team in a sense that they can decide whatever it seems fit.

Now when it comes to errors, they need robust methodology to tackle the errors occurred and a plan so that it doesn't ever happen again. Their immediate step was the following:

- Reboot servers
- Disable automation
- Check rest of the site's status
- look for config changes or deployment

These steps are simple enough to tackle errors occurred followed by small hack or fix to keep the site running for the time being. After this software engineers gather together in "War room". Where they try to do the following:

- Try to keep the rest of the site running
- Reproduce failures on a small scale
- Look for code/config changes
- Dig into and analyse data

Now, this is the immediate measures taken by the engineers at Facebook. But since we need something to fix the problem such so that it does not happen again. For this they do the following:

- Focus on the root cause (Deep dive and Not blame)
- Find how to prevent recurrence
- Do not rely on best intentions

- Always remember humans are desperately unreliable

They showed with example errors that actually occurred on Facebook and how they followed the above steps and guidelines to fix the memory leak and overloading of traffic on servers problems.

To prevent failures unit testing, integration testing and load testing are one of the ways. But testing would not catch all the failure nodes and one can't test an entire distributed system at scale. Hence, they try to:

- Roll out stuff slowly
- Monitor KPIs
- Fail fast than timeouts
- Defensive servers

Since contribution, changes, code and bugs are all proportional to each other. Hence, small changes help them to figure out what broke when. The error tackling process is very different from IBM's as most of the errors are caught before the development pipeline and if not immediate fix is created but proper fix is done in the later stages of the cycle of development whereas in Facebook they do not of have cycle of development, they release as they build and if error occurs they rollback and fix and then keep developing.

The code rolling out is in small chunks but every day. Whereas in IBM's agile process rolling out of new code is every 2-4 weeks. Every team in IBM follows a proper agile planning whereas in Facebook it is on ownership basis. Testing and code reviews are done by both the companies in the same way but when it comes to testing IBM have their own pipeline of testing where it goes through a number of nodes (Testing on local, Testing on a server, UAT testing) before declaring as pass. Whereas in Facebook, test are usually against what they develop and only those test and their dependencies are run.

I feel such varied in the development process is due to the sheer number of computing and data. If Facebook had to use IBM's way it would fail because rolling out such big change is risky. Rolling out to such large data centres is another impossible task. With such large code base testing, each of them would take ages. And because of this sheer number they do something call reactive programming where they automate most of the tasks and wait for the signal from the system if something is wrong. They both have automated tools for detecting software vulnerabilities, code smell etc. Facebook uses their own whereas IBM uses SonarQube.

If the same is reversed where IBM uses Facebook way of development it might not do as well as IBM has way more employees than Facebook and managing and tracking would become a difficult task. Also, most of the Facebook's apps run on their servers, hence they provide services. Whereas IBM does both they provide services by running software on their machines and they also provide software to be run and used by the buyer. Hence rolling out would not be possible every now and then. Also, rigorous testing is required as rolling back or updating a software running on someone else's machine is difficult and inconvenient. So, for making sure that their rollout is stable their software has to take an agile process of development.

7.11 Q & A

Some question that were asked at the end of seminar were:

1. How do they manage the test as their code base is huge?
   Answer: They focus more on unit testing as they are quite fast. They have tools which run tests only on the new code and their dependencies.

2. Development methodologies used at Facebook?
   Answer: Development process is chosen by the team. They choose the best model according to the project.

3. How is code quality maintained in such a large codebase?
   Answer: They have automated tools to find code smells and memory leaks. Also code review is a must before going into production.

7.12 FoodCloud seminar

7.13 Speaker

Roy Phillips, Chief Technical Officer at FoodCloud.

7.14 Report

The speaker started off with explaining about FoodCloud. FoodCloud has deals with the various grocery stores. When grocery stores cannot sell perfectly good food. They upload the description of the food using their in-store scanner or their smartphone app. FoodCloud has many links with the local charity. They let the local charity know about the unsold food that the grocery stores couldn't sell. The charity can respond to the notification by accepting and collecting it.

They not only have client and charity app, they have text notification service, a website and have various integration with their clients. The toughest part in FoodCloud is the integration of their platform with their clients i.e Tesco etc. We can see that IBM and Facebook mostly deal with providing services that they usually make from scratch and run on their own servers. Integration is usually one of their tasks but mostly they are the providers providing the services. FoodCloud, on the other hand, acts as a middleman where they create a bridge between the grocery stores and the charity. This requires software to be very versatile and flexible to be able to handle integration with ease.

Facebook did not seem to give much care to the software tools or platform they used. IBM mostly used Java for creating programs. FoodCloud uses Scala which addresses the criticisms of Java. Scala also uses Java VM. This is due to the robustness and cross-platform Java that the FoodCloud chose to use. FoodCloud has to integrate with multiple clients hence they do require cross-platform scalable tools to be used.

In regards to the running of the application, Facebook and IBM have their own big data centres as they are such large companies. Whereas since FoodClous is a growing non-profit business, they need a scalable solution. For this reason, FoodCloud chose Heroku for deploying the apps for its ease of deploying and scalable plus they also are expandable based on needs meaning they only have to pay for what they use.

Tracking of the progress of the software development was done using JIRA. They employed simple backlog mechanism where they pick the most important tasks and work on it. Tracking the task in the backlog. They followed Kanban and Agile methodology. The speaker did not believe in estimating the time required to finish the task. As in software development, it's very hard to estimate how much time it is gonna take to complete a task. Wasting time on estimation is hence unnecessary. This ideology is quite valid in real life. Spikes are very common in software development. Spike is essentially when trying to finish a task, the developers find that this task actually requires major work and more time than the estimation. These spikes tend to obstruct the agile process. Hence the speaker followed a methodology where his team took the maximum of 4 stories and worked on them.

The speaker strongly followed the SOLID principles. SOLID principles are the design principles in OOPs. FoodCloud used technology and designs such as SRP, Akka Streams, RabbitMQ for messaging etc. Testing was given a lot of importance by the speaker. I feel the technologies chosen by the speaker are indeed quite robust and scalable. The hosting on Heroku does make sense as the want minimize their cost. They follow continuous integration and development.

I feel that the process and the tools are well suited and chosen for the development of FoodCloud. It is interesting to see different methodologies and process used by various different companies. The number of projects, the size of the project and the type of the project drive the type and methodology chosen. IBM used one process to manage different teams namely agile methodology. Whereas Facebook did not care much about the process and gave the developers the freedom to choose their methodology. It is interesting that Roy takes a different approach quite similar to Facebook. Rather than following strict methodology they tend to focus on simple backlog like implementation. This is due to the sheer size of users and the size of the company. Another reason for choosing so that I believe is that FoodCloud requires them to interact with various developers of the client the company is working with to integrate their system.

7.15 Comparison of Company Seminars

Bibliography