Level 3 Project Case Study Dissertation

SH24 Team

Gianmarco Cornacchia
Daniel Flynn
Luke Mullen
Alaa Wardeh
Yuqi Li

7 April 2023

Abstract

This paper is a case study on the development of the team allocation web app developed for Barclays by team SH24 as part of the Computing Science third year team project at the University of Glasgow.

Upon outlining the background of the project, the paper reflects on wider concepts in the areas of software development, teamwork and project management in order to analyse the causes and circumstances of various events and incidents that characterised the development journey.

Finally, it further reflects on the insights gained by the team while developing the software solution, discussing their general relevance in a software team setting.

1 Introduction

Throughout the case study we will be analysing the development of the team allocation Django web app that was developed by team SH24 in order to automate the allocation of graduates to new teams in the Barclay's graduate rotation process.

The paper starts off outlining the background of both our customer and the project we were tasked to carry out, specifying our project aims, and what we were able to achieve. Afterwards, the case study moves on to state our reflections on various aspects of our project, ranging from human factors like team communication to more technical matters such as the use of version control, branching strategies and quality assurance. The paper then moves on to reflecting upon unclear project requirements, ending the reflection with a discussion about the challenges that the team faced especially on the grounds of reduced manpower and a lack of senior guidance.

The rest of the case study is structured as follows:
Section 2 covers the case study background. Section 3 delves into communication issues within the team, highlighting their causes and effects on team productivity. In Section 4, we discuss our software process. In Section 5, we describe our approach to unclear requirements, moving on to Section 6, where we reflect upon how we handled a team member dropping out and guidance issues.

2 Case Study Background

We were assigned the task of automatically allocating new teams for members of the Barclays graduate rotation program as they transition from one team to another.

2.1 Customer Background

Barclays is a British multinational investment bank and financial services company, headquartered in London. Operating in over 40 countries and with strong presence in Europe, Africa and the Americas, Barclays is one of the largest banks in the world. The company offers a range of financial products and services, including retail, commercial and investment banking, wealth management, and credit cards [4].

Early on in the project, our point of contact at Barclays made it clear to us that our product would not see use in a production environment at Barclays. The primary reason for this was security concerns regarding the handling of confidential employee information. The security policies and guarantees required for software used at a company with the nature and magnitude of Barclays was both beyond our expertise to implement, and beyond our means to insure. Instead, the customer wanted a proof-of-concept solution for automating team allocations, being particularly interested in our initial approach to the problem as a team of relatively inexperienced developers, and in the way our process would refine and improve throughout the project.

2.2 Project Aims

As mentioned in section 2.1, the customer was mainly interested in what our approach would be to an ambiguous task with very few initial requirements. This was reflected in both the initial project outline and our first meeting with the customer; the former comprised only a single sentence outlining the problem and its premise, and during the latter we were told that other than the must have requirement of automated allocation, there was no strict guideline as to how to solve the task. The implication of broad requirements is discussed further in section 5.

We decided the most appropriate form of software for this task was a web app; some of the motivators behind the choice being that the large user-base of graduates would be able to easily access the software due to cross-platform compatibility [7], the use of external consistency principles for user interface design would make software use intuitive [18], and a web app could support uploading and serving data from multiple machines. We drafted an initial set of requirements using the MoSCoW system [17], which we presented and discussed with the customer on the first customer day, then refining them throughout the project. The outline of the resulting requirements was a web app with support for three user types: 'graduate', 'manager', and 'admin'. On the 'admin' side the web app would support the upload of user and team information, the ability to run an allocation algorithm, and the ability to extract the results. On the 'graduate' side, the web app would support submitting preferences pre-allocation, and viewing the results post-allocation. Finally, a 'manager' would be able to see graduates allocated to their team and modify team data.

2.3 Achievements

We delivered the code base for a fully functional web app which automated the allocation of members of the graduate program to new teams, as well as a live demonstration site the customer could access. The only non-automated part of the web app's workflow is the configuration of an admin account login details, which must be done in a terminal when the web app is started.

2.3.1 Technical Details

The app was developed using the Django web-framework for Python, and the front end was built using a combination of HTML and JavaScript. The database engine used was SQLite, for which code is generated and executed using the 'makemigrations' and 'migrate' commands built in to Django. As the app's purpose was not to be deployed, a file-based email system was used to emulate normal email functionality. Instructions were included in the 'README.md' regarding how the app can be configured for live use with a mail server. A live demo for the app was hosted using 'PythonAnywhere' (`https://sh24.eu.pythonanywhere.com`), and instructions for use were included in the handover document provided to the customer following our agreement on the handover process.

2.3.2 Admin Features

Instructions for configuring the login details for an admin account are included in the project's 'README.md', in the root directory of the project's repository. Once the account is configured, an admin will be able to login through the web app's login page. When logged-in, they are presented with a portal from which they can navigate around the site. The portal also contains information on how to format CSVs of user and team data. The portal contains buttons which link to two upload pages, where the user can upload CSVs containing user and team data. These CSVs are used to automatically populate the app's database user and team models. The portal also contains links to pages where an admin can manually add users and teams. Once the database is populated, a button is enabled that allows the admin to run an allocation algorithm, which places graduates into teams depending on their preferences. A new page is displayed which shows the admin all the teams and allocated graduates. In this page the admin can manually change team details, including adding, removing, and exchanging graduates between teams. The page also includes a button to download team allocations in a CSV format, allowing the allocation result data to exported in the same format as the user and team input data.

2.3.3 Graduate Features

When a graduate account is created by an admin, the email associated in the account will receive a password reset link, which they can use to setup their initial login details. A forgot password functionality is found on the login page if the password is forgotten or needs to be changed. When logged-in, before the allocation has been run, the graduate is presented with all the teams available to them. They can click on the teams to view a drop-down display of teams' details, including each team's department, manager, capacity, and description. The graduate can express their preference by 'thumbs upping' teams, up to a limit of five 'thumbs up'. Once the admin runs the allocation, the graduate will receive an email notifying they can view their new team. When they next log in to the site, they are presented with a page displaying the details of their new team, including the other graduates allocated to the team.

2.3.4 Manager Features

The initial configuration of login details is the same as for a graduate. When logged-in before the allocation is run, the user is told to return when notified as a manager does not have any use for the web app in that state. When the allocation is run, the manager also receives an email notifying them. Once logged-in, they can view the details of the teams they manage, manually remove graduates, and manually add graduates who do not currently have a team.

2.3.5 Allocation Algorithm Features

We approached the allocation problem as a combinatorial optimisation problem, where the optimal solution would be an allocation where every graduate was assigned to their most preferred team possible. We consulted an expert at the University of Glasgow, Dr David Manlove, who advised us to model the problem as a 'minimum-cost flow problem' [16]. He also advised us that no optimised solution existed for a problem such as ours, where the flow in the network would not necessarily be zero, as the total capacities of teams may be larger than the total number of graduates [16].

Taking heed of this advice, we modelled the problem as a bipartite flow network. The first set of nodes were the graduates, which were all sources emitting "1" flow into the network. The second set of nodes were the teams, which were sinks receiving flow equal to their capacity. The cost of flow traversing an edge was equal to the inverted preference value of the source graduate node for the sink team node (e.g., if a graduate "thumbs upped" a team five times, indicating highest preference, the cost of traversing the edge between their nodes would be minimal, i.e., "1"). The capacities of edges did not matter as long as they were greater than or equal to one, as the largest flow between a source and a sink was 1. We used the 'min_cost_max_flow' function from the NetworkX python package to calculate the optimal solution to our model. This function implements the network simplex algorithm, the most efficient method for solving "minimum-cost flow problems".

However, as mentioned before, this optimised solution only existed for flow networks with a total flow of zero, and that was unlikely to be true for the majority of our use cases; more often that not there would more total vacancies than graduates. We overcame this issue by implementing a two-step approach to our allocation algorithm. In the first step, a randomly sampled set of graduates is selected, where the size of the sample is equal to the total minimum capacities the teams require. In the second step, the remaining vacancies in teams are proportionally shrunk until the number of total vacancies is equal to the number of remaining graduates.

3 Team Communication

In this section, we reflect on the communication problems that arose during our team effort. For most team members the third year team project was the first opportunity to engage in a large-scale software team project. Through discussion and reflection during our retrospectives, we realised that cooperation among the different individuals in the team wasn't always as effective as it should have been.

One of the main discussion points during our first retrospective was about how often people wouldn't reply in our Microsoft Teams chat. Using Taiichi Ohno's 5 Whys technique [19], we understood that the root of the problem lied in the inconsistent notifications on Teams. Therefore, we decided to create a WhatsApp group chat that was mainly used to remind people to check Teams when a message was sent. Furthermore, at the start of the project we hadn't set a specific time when we would meet on our lab day. This caused last minute coordination every week, causing major time inefficiency. Conscious that coordinating activities can take up around 20% of our time [10], we decided to set a fixed meeting time, agreeing that anyone joining the group late would need to bring snacks for everyone else; a light-hearted way to incentivise people to show up on time.

Additionally, one of the team members experienced a considerable language barrier while communicating with the rest of the team. In order to truly work as a team and keep everyone on the same page, we have taken the following measures as a team: Firstly, we tried to speak slowly, using a simple vocabulary [9], secondly, we made sure to write down the main talking points of any meeting to ensure understanding [9] and thirdly, we promised to be honest to one another and ask for clarification when needed.

During our retrospective at the end of the third iteration, we noticed that it was common for team members to be unfamiliar with the software developed by other people, highlighting another problem in team communication. We quickly realised that we had to establish an "effective communication network" [1] to ensure the participation of every team member. Based on the Agile software development methodology, we decided to tackle this problem by having weekly stand-up meetings at the start of our weekly working day. Stand-ups enabled us to "bring everyone up to date on the information that is vital for coordination" [2], thus ensuring more balanced contributions and accelerating our development progress.

4 Software Process

4.1 Effective use of version control

Having a consistent, efficient, and effective software process has been one of the team's priorities since the start. The first challenge that arose at the very start of the project was to make sure that everyone was on the same page in terms of development practices. For most Computing Science students at the University of Glasgow, the third year team project is the first real opportunity to work on a large-scale software project. Having had experience in programming for several years before, all the personal workflows, conventions, and processes developed when working mainly on individual assignments may clash when working as a team.

One of the best examples of this challenge is the variable expertise in the use of version control systems like Git. As confirmed by the 2016 GitLab Global Developer Survey, "the biggest concern respondents cite about using Git is the associated learning curve, with 40% saying they consider it an issue" [23]. However, overcoming the Git learning curve is essential to a successful team project. That is because it allows team members to work with different versions of the same project, making it easy to "base new work off any version of code" [14], and synchronizing each version to "make sure that changes don't conflict with changes from others" [14]. Furthermore, it enables a team to "keep a history of changes as the team saves new versions of the code" [14], allowing to keep an always-updating history of the project.

In our specific case, we made sure that everyone understood the importance of properly using Git during our first team meetings. However, we quickly realized that everyone would have benefited from a workflow guide that could be followed when developing. This would have allowed everyone to follow the exact same process and would have directed everyone to the same resource when one forgot the proper workflow. Therefore, we created a GitLab wiki document called "Git Workflow and Conventions". This document outlined all the different steps of a proper Git Workflow, from creating and checking out a branch to committing and pushing code to the GitLab remote repository. In this way, the members of the team that were less experienced in the use of version control systems had a simple guide that they could easily follow in times of need.

4.2 Feature Branching

Included in the "Git Workflow and Conventions" guide, we also outlined our development strategy in terms of branching. We decided to follow a feature branching strategy, thus having all "feature development taking place in a dedicated branch instead of the main branch" [22]. In practice, every developer would open a new branch from the GitLab issue corresponding to the feature that was going to be developed and worked on the feature on the newly created branch. We chose this strategy because we believed it was the most convenient and hassle-free way to work on a large-scale project as a team. In fact, the feature encapsulation that is guaranteed by this workflow "makes it easy for multiple developers to work on a particular feature without disturbing the main codebase", making sure that each feature is included in the main branch only at its final and refined stage through a merge request.

Feature branching proved to be very effective at managing the constant concurrent change of the codebase, keeping merge conflicts at a minimum and allowing us to follow a logical and encapsulated progression. However, this strategy had its drawbacks as well, which became apparent to us throughout its use and were mainly discussed during our retrospective held at the end of the third iteration.

Firstly, we realized that with the biggest features of the webapp being already developed, smaller changes to our codebase were becoming more frequent, driven primarily by issues concerned with bug-fixing and feature refinement. This meant that often we would create a branch, develop a very small feature, and then merge the branch into main, with the associated code review and conflict-resolving overhead. Being these features small and quick to develop, we found ourselves having many branches and merge requests, which "take time (and people) to manage" [12]. Therefore, we decided we would only use feature branching for bigger features and make ad-hoc branches that would include several similar mini-features or bug-fixes in order to increase efficiency and reduce the overhead.

Another issue that we found with a strict feature-branching strategy was that it relied on people being very careful and diligent with changing their current local branch when developing on different features. Especially in times close to feature deadlines and customer days, our team would be so immersed in the development of various features, that it was relatively easy to forget to switch branch and pushing changes to the wrong one. This happened a few times, decreasing the modularity and encapsulation of our version control, and increasing confusion in terms of issue management and code reviewing. During the retrospective we did not succeed in finding an action that would specifically tackle this problem, however, we agreed to be more careful and stricter about this issue, and to keep each other accountable on this matter.

4.3 Code Reviews

Following feature branching made it easy for us to incorporate code reviews into our development process. The merge requests happening at the end of each feature's development represented the perfect opportunity to review someone's changes and either approve or make comments to suggest various changes.

For the whole duration of the project we assigned someone else as a reviewer for each merge request, however, especially in the busiest sprints in terms of development, we understood that we were not being thorough enough with our code reviews. "The trade-off is between effective use of engineers' time and maintaining code quality" [11]. We had realized that our balance was more shifted towards the effective use of our time, thus decreasing our overall code quality. We solved this by spending more time on code reviews, and we quickly saw our code quality improving, following the advice given by reviewers during each merge request.

Another issue that we found with code reviews was strictly tied to the issue that was mentioned in section 4.2 regarding the high number of branches that was being maintained. Since each branch's existence will end with a merge request, to a large amount of branches corresponded the same amount of code-reviews. This factor caused our team's development to slow down at times mainly for two reasons. Firstly, code reviews took a toll on the developers' time, which could not be used on developing new features, and secondly, if a feature was dependent on one that had not been merged into the main branch yet, the only ways to progress to the next feature would have been either developing on the wrong branch, our pulling an unreviewed branch into a new one, both of which would be considered bad practices.

5 Unclear Requirements

As hinted in section 2, one of the most persistent challenges we faced was due to the software requirements being quite unclear and changing at times. For example, perhaps the most impactful requirement was which type of deliverable we should bring to the customer at the end, if it should be a web app, mobile app, desktop app, etc. We were simply told to decide on this amongst ourselves, and whilst this had the advantage of giving us more control over the project, using software platforms we were more comfortable with, it also used up valuable time which could have gone towards design and development. After deciding that a web app would be the most applicable format for an application like this, due to the ambiguity of the requirements, when it came to picking exactly which web framework, we spent a further week or two deciding on whether to use ReactJS or Django, before finally settling for the latter. Once we had a deliverable chosen, crucial time had passed and we found ourselves quite behind in terms of development progress.

What we found from the customer was that the requirements unfolded gradually over time. We found ourselves having to go back to the customer on each meeting with our interpretation of their requirements and see what their feedback on that was (to which, it was usually positive). One such case was the customer specifying, in a subsequent meeting, that allocations should only be executed after all of the preferences have been cast, rather than allowing further preferences to be cast after an initial allocation has been run, perhaps leading into the possibility for multiple executions of the allocation function. Looking back, this was a massive change in product functionality and could have certainly changed the overall direction of the project. However, at the time it did not appear to be such an issue. Another example of this relates to the manager's instance of our application. Our customer only notified us of their desire for this functionality on the second meeting, once design had already been started. Furthermore, the specifics as to what should have been included within this section was left mostly up to us to figure out, beyond the vague requirement of a manager being able to "look at the details for their specific team".

During our initial meeting, the minimum viable product was simply "a system that performs automatic team allocation" and we were given no strict requirements on how to actually solve the problem. This unsurprisingly led to frustration during the design of the algorithm for the application. First, we were told that there would be around fifty graduates per rotation, but were also told to keep the project as generic as possible for potential use by other corporations – so were we to build an algorithm that was optimised for 50 or 50M+ people? Next, Professor David Manlove let us know that another crucial piece of information to a problem such as this is knowing the lower-bound of each team (that is, the minimum number of members that may be in any one team) – so crucial that depending on the number, it could bring us into the realm of NP-completeness [16]. On reflection, this is clearly an important part of the problem which, whilst we received the information eventually from our customer, would have sped up and simplified progress if we had known it from the start. Unsurprisingly, having clear requirements from the start has the potential to massively speed up software progress, and a lack of them can "undoubtedly delay

[the] project further" [20] . Unfortunately, this is indeed what we experienced in this scenario.

Why might our customer have deliberately made the requirements ambiguous anyway? It is interesting that "the most common of the experienced consequences [of poor-quality requirements] ... was extra communication" [15, p. 45]. In retrospect, it did indeed cause us to communicate more effectively with one another, as on numerous occasions we would have conflicting interpretations within the team needing resolved, due to the poor requirements. Furthermore, there exists the case where a developer and tester have different interpretations, and so when it comes to testing, leads to "the developer having to re-do their work" [15, p. 45]. This indeed happened to us when one developer implemented a "downvote" button on the site, only for another to note afterwards that this functionality was not needed. This was frustrating for both parties, and was a waste of time – was this also an intended outcome our customer had? We believe not. Instead, it was an unfortunate drawback of the ambiguity. Overall, it would make most sense if the customer was trying to probe us into communicating more with one another, as whether this was on friendly or hostile grounds (it had been both), it would prepare us well for real-life, going into a work environment where this is the norm.

It is a valid question to ask: why did we not just go back to the customer and clear up the ambiguity? It has been said that "one of the best ways to take ambiguous business requirements and turn them into quality software is by pushing back on them and clearing up any misunderstandings" [3]. We did find ourselves probing the customer about the requirements on each subsequent meeting, but found it challenging since we were always hyper-focused on singular things, leaving the meeting realising that we had forgotten to clarify other areas. Looking back, we would certainly do this differently if we were to do it again. We would follow advice, such as that outlined by Anderson [3], and attempt to clarify any ambiguity earlier on. Then, we would try and clarify as much as possible, even about areas of the project which we have not begun design or development on yet, so we have a better idea earlier in the timeline. Above all, we did make the best of the situation given a customer who was persistent in giving unclear requirements and see both the positives and negatives of this.

6 Team member drop-out and lack of coach support

One of the first major issues we had to work through in this project was losing one of our teammates during the beginning of the year. We had started off as a group of 6, and everyone was attending regularly and consistently, and communicating and working as expected. However, one day one of our teammates, who also happened to be one of the most experienced among us all, left the team without any contact or explanation of how to pick up where he left us off. This left us with a big gap to fill, especially given his degree of experience with many technologies, which made him often a key member in our discussions.

To make matters even more difficult, he had not given us any final guidance or advice on how to continue with where he left off on his portion of the work on the group project, meaning we also had to work together to figure out exactly how to carry on with his portion of the workload without being able to refer to him for any clarification. Most of the existing literature on issues like this talk about communication with the leaving team member as the key factor to keep succeeding [6], however in our case we had no way of communicating with him, as he was not replying to our messages.

One of the biggest setbacks we got from this incident was that he had insisted on developing the front end using React. He was already experienced with that platform and told us that he would take on most of the front end workload as a result. We agreed with him despite none of us having any React experience as he seemed keen and capable of handling it as he promised. However, once he had left, we had no one who could pick up the React work straight away. At first a couple of us tried to learn React, thus putting front end development on hold for a few weeks. We asked ourselves, "What's the long-term solution? Can your current team get the project done?" [5]. Given the fairly steep learning curve, and seeing we had already been held back a few weeks, we agreed to switch to a full-stack Django approach, which we all had some experience with.

Another major struggle which we had to constantly work around was that we did not have a coach to support us throughout the majority of our project. Every team was assigned a coach since the beginning, whose job was to meet with their corresponding team during the scheduled Wednesday meeting slot in order to provide guidance and engage with the students in various software development exercises. Their guidance is especially vital given the importance of the software process in our project, where, as more experienced students, they can help the team improve their development approach in various ways.

Our struggles with the coach situation started early on in the year when we were given our original coach. We had our initial meeting with him and everything went well, he provided us with helpful advice and answered all our questions thoroughly. However, we were told shortly after our initial meeting that our coach had to drop out and that we would be receiving a new one.

After waiting a few weeks without a coach and following up on the situation several times, we were assigned a tutor to act as our coach, which meant he did not have the training that regular coaches undergo. By this point in our development timeline, we had already decided we wanted to use Django to develop our web app and had already started working on the high-level design of our product. This caused everyone to feel like we were out of sync with our coach, as we had already gone through the entire design stage and had started on the actual programming without our coach having even been assigned to us. He was also busier and thus more difficult to get a hold of and we were not able to meet him every week as expected, which made this issue even more difficult to work around.

The lack of a guiding figure to help us work through this large scale project of completely new territory was a big challenge for us to work around. We mainly tackled this challenge by voluntarily reaching out to more experienced individuals for guidance. We spoke to our marker several times, who gave us great insight on how to make the most of our customer meetings. Then we reached out to David Manlove, who was tremendously helpful in our matching algorithm design stage, and finally we spoke to the course coordinator a few times in order to make sure we understood how we were being assessed. Overall, despite having objective guidance issues that were outside our control, we were able to work together and make use of all the other support we had around us to get the guidance we needed to complete this project.

7 Conclusions

For everyone in the team, this project represented our first large-scale software project with a real customer. This gave a novel level of depth and importance to our project, which was far different from any other assignment that we were tasked to do as part of our Computing Science degree. With a real customer came real requirements, deadlines and expectations, which we were able to manage well by delivering a product that our customer was satisfied with in the specified timeframe. This was made possible by a group of individuals that was eager to learn new concepts, technologies, processes and wider lessons that were encountered along the way.

As a team, we learned to communicate effectively with one another, making sure everyone was on the same page about development choices and progress. Agile ceremonies like stand-up meetings helped massively by enabling us to give everyone else an overview of the work done during the past week. Evaluating other case studies such as the one carried our by Viktoria Stray [21], analysing the pros and cons of stand-up meetings, our team setting and the nature of our work enabled us to make the most of the positives, without experiencing any negative aspects.

Regarding software processes, having a ”common working framework” [13] is highly regarded by existing software engineering literature. We tried to achieve that by learning about new processes together and by writing down our team's common workflow on our GitLab wiki. This created a common ground regarding important practices such as the use of version control, issue management, and branching strategy. Furthermore, we learned to focus more on quality assurance in the later stages of the project through testing and code reviews, which, in retrospect, we should have implemented since the very start. If we were to start the project again, we would definitely try out new software engineering practices like Test Driven Development, in order to keep our code clean and tested since the very start.

Working with unclear requirements is probably one of the most common problems in software projects. In hindsight we should have understood how this was going to affect our progress earlier on, in order to communicate better with the customer and amongst ourselves since the start. It took a while for us to be able to effectively narrow down a list of requirements and work towards them, but we were able to successfully achieve this in the end. In retrospect, looking at the positive side, we got the chance to work in conditions that cause many software projects to fail [8], learning ways to mitigate these circumstances and succeed in future projects that may be impacted by the same issue.

Finally, with a team member dropping out, we learnt the importance of truly working as a team in order to be affected as little as possible by such incidents. We understood that when our teammate left, we were not fully aware of his development progress and vision, thus setting us back for weeks as was previously discussed. Furthermore, given the absence of a coach for most of the project, we learnt to look for guidance on our own, learning to openly ask for help when needed.

Overall, everyone in the team learned valuable lessons that will be applicable to both software and non-software related projects in our future. We learned to tackle problems that often impact the software engineering industry, thus bettering our chances of success in our future endeavours.

References

[1] Stephanie Adams. Building successful student teams in the engineering classroom. https://www.researchgate.net/publication/238696668_Building_Successful_Student_Teams_in_the_Engineering_Classroom, 01 2003.

[2] Agile Alliance. Daily meeting. https://www.agilealliance.org/glossary/daily-meeting/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'daily*20meeting))~searchTerm~'~sort~false~sortDirection~'asc~page~1).

[3] Russell J. Anderson. Ambiguous business requirements. https://russelljanderson.com/ambiguous-requirements/, 02 2018.

[4] Barclays. Who we are. https://home.barclays/who-we-are/.

[5] Base36. How to handle a key team member leaving at a critical project point. http://www.base36.com/2013/02/how-to-handle-a-key-team-member-leaving-at-a-critical-project-point/.

[6] Deanna DeBara. What to do when an important team member quits or goes on leave. https://blog.trello.com/when-a-team-member-quits-or-goes-on-leave, 07 2021.

[7] Jemin Desais. Web application vs. desktop application: Pros and cons. https://positiwise.com/blog/web-application-vs-desktop-application-pros-and-cons/.

[8] Azham Hussain et al. The role of requirements in the success or failure of software projects. https://www.researchgate.net/publication/308972993_The_Role_of_Requirements_in_the_Success_or_Failure_of_Software_Projects, 10 2016.

[9] Dennis Medved et al. Challenges in teaching international students: Group separation, language barriers and culture differences. https://www.semanticscholar.org/paper/Challenges-in-teaching-international-students%3A-and-Medved-Franco/7e1892eb441110afe9f546dfd2a7d899b7e375a4, 10 2013.

[10] Gary M. Olson et al. Small group design meetings: An analysis of collaboration. https://www.researchgate.net/publication/232905224_Small_Group_Design_Meetings_An_Analysis_of_Collaboration, 12 1992.

[11] Robert Fink. Code review best practices. https://blog.palantir.com/code-review-best-practices-19e02780015f, 03 2018.

[12] Brad Hart. Trunk based development or feature driven development — what's better for your team? https://www.perforce.com/blog/vcs/trunk-based-development-or-feature-driven-development#what-trunk, 10 2019.

[13] Watts S. Humphrey. The team software process (tsp). https://apps.dtic.mil/sti/citations/ADA387279, 11 2000.

[14] Microsoft Learn. What is version control? https://learn.microsoft.com/en-us/devops/develop/git/what-is-version-control, 11 2022.

[15] Emil Lund. The impact that the quality of requirements can have on the work and well-being of practitioners in software development. https://www.diva-portal.org/smash/get/diva2:1637596/FULLTEXT02, 2022.

[16] David Manlove. Personal Correspondence via Microsoft Teams, 26-10-2022 to 20-03-2023.

[17] Eduardo Miranda. Moscow rules: A quantitative exposé (accepted for presentation at xp2022). https://www.researchgate.net/publication/356836488_MoSCoW_Rules_A_quantitative_expose_Accepted_for_presentation_at_XP2022, 12 2021.

[18] Jakob Nielsen. 10 usability heuristics for user interface design. https://www.nngroup.com/articles/ten-usability-heuristics/, 11 2020.

[19] Taiichi Ohno. Ask 'why' five times about every matter. https://web.archive.org/web/20221127052017/https://www.toyota-myanmar.com/about-toyota/toyota-traditions/quality/ask-why-five-times-about-every-matter, 03 2006.

[20] Damian Scalerandi. Want to speed up software development? https://www.bairesdev.com/blog/want-to-speed-up-software-development/.

[21] Viktoria Stray. An empirical investigation of the daily stand-up meeting in agile software development projects. https://www.researchgate.net/publication/266850154_An_Empirical_Investigation_of_the_Daily_Stand-Up_Meeting_in_Agile_Software_Development_Projects, 08 2014.

[22] Atlassian Tutorials. Git feature branch workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow.

[23] Emily von Hoffmann. Why git is worth the learning curve. https://about.gitlab.com/blog/2017/05/17/learning-curve-is-the-biggest-challenge-developers-face-with-git/, 05 2017.