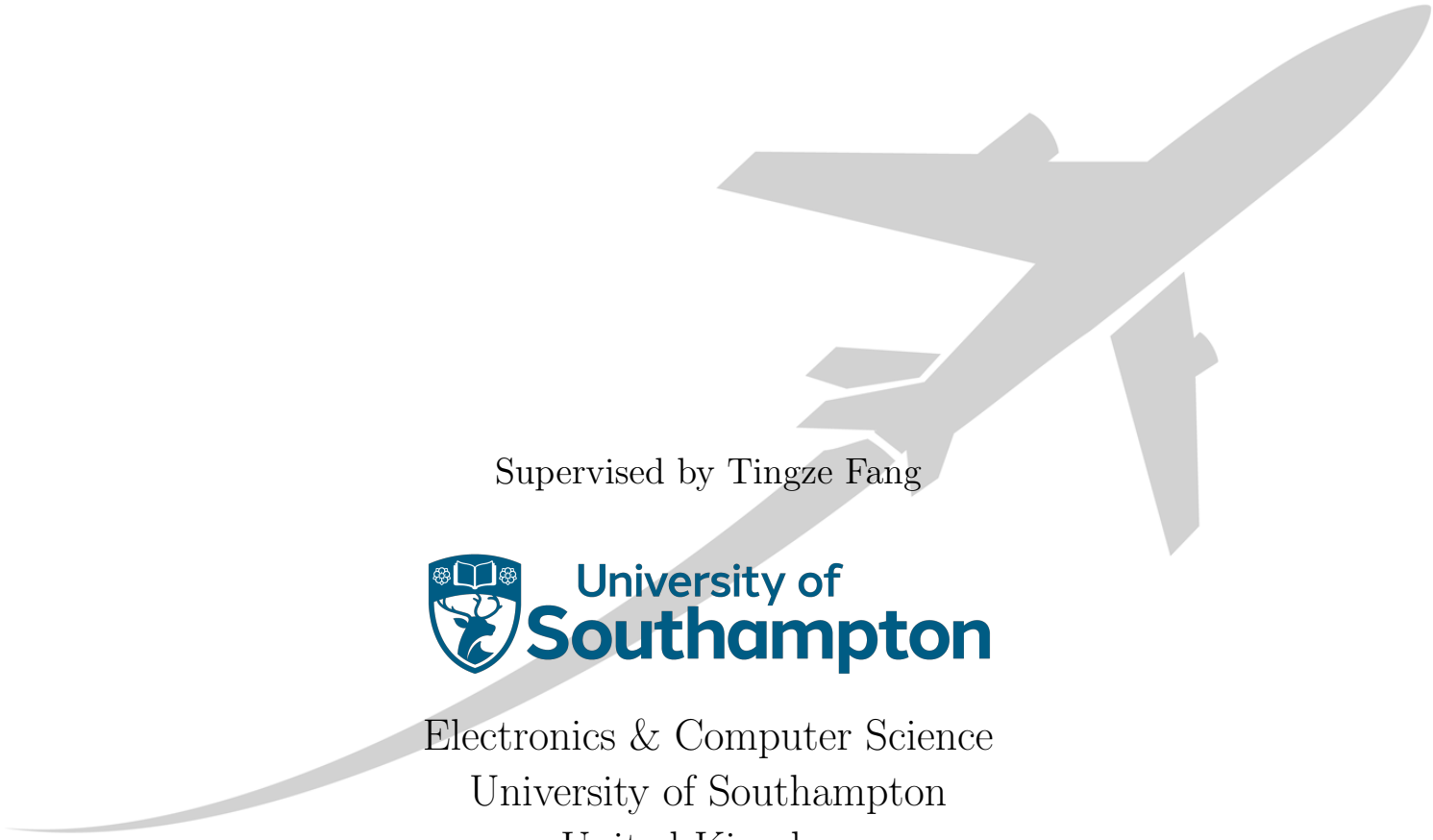
COMP2211
Software Engineering Group Project

Runway Re-declaration
Deliverable 5: Final Report

Group 45

B. Jegatheeswaran (bj4g22@soton.ac.uk)
A. Geron (ag7g22@soton.ac.uk)
Y. Balasubramaniam (yb2e20@soton.ac.uk)
M. Gu (mg2n22@soton.ac.uk)
S. Zagrosi (sz10g21@soton.ac.uk)



Supervised by Tingze Fang

Electronics & Computer Science
University of Southampton
United Kingdom

Contents

1 Introduction

This final report documents our team's semester-long journey in developing a Runway Re-declaration Tool for the Software Engineering Group Project. Throughout the project, we applied various methodologies to manage our work effectively. Regular communication allowed continuous improvement and ensured our application met project requirements. Our team collaborated incredibly well, leveraging each member's individual strengths to complete the project to a standard that exceeded our own expectations. Despite some challenges which this report will explore in more detail, we delivered a high-quality application exceeding the specification-required functionalities.

We aim to provide an in-depth evaluation of our team's performance, resource management, and work efficiency throughout the project. Moreover to assess the tools and methodologies employed, evaluating their effectiveness for this project. By reflecting on our successes, challenges, and lessons learned, we also aim to provide valuable, transferable insight for future students working on the Software Engineering Group Project.

2 Evaluation of Team Work

2.1 What Went Well

Our team worked well together, managed time as effectively as possible, and used agile methods to the best of our ability. Thorough planning in the initial stages helped us stay on track during each sprint. We completed many of our most difficult core features (Must and Should user stories) before Sprint 3 giving us time to add extra features.

Using Scrum was helpful, and our meeting channels enabled effective communication and project improvement. We had regular updates about progress, ensured everyone felt comfortable seeking help, and each individually contributed to our shared success. We aimed to commit feature updates regularly to uphold progress consistency. Our respectful and transparent group culture allowed us to maximise the use of each member's strengths and support their weaknesses. By encouraging self-organisation and cross-functional cooperation, we leveraged our diverse skills and perspectives. This improved our problem-solving capacity when dealing with difficult user stories and created a sense of shared ownership and accountability for the project's success. For instance, we utilized team members' expertise in areas like GUI development to create a visually appealing application, while relying on others for their technical skills in back-end development to ensure that no one felt too out of depth with their tasks.

Using DevOps practices like continuous integration and automated testing improved our efficiency and code quality by setting concrete expectations for code behaviour, catching issues early, and thus reducing technical debt. For example the use of XP principles such as pair programming, in which two developers would work together on the same task, to ensure we stuck to specifications and produced better code. We rotated pairs across user stories and increments to encourage collaboration and learning among all team members. Continuous integration by frequently merging our code changes and running automated tests was a priority.

2.2 Areas for Improvement

Despite the overall success of our project, we identified several areas for improvement in hindsight. We anticipated some problems as nobody in our group had worked on a collaborative project of this scale before. While we adopted agile practices, there was room for improvement in our adherence to some of its principles. Nonetheless, mistakes in our approach were varied both in symptom and time-cost. Through detailed analysis and reflection, we aim to learn from these errors to improve efficiency in future projects.

A major source of grief was our version control methodology using GitHub. Our disorganised one-branch approach led to frequent merge conflicts, often requiring many hours to resolve. We had instances in which the main branch had to be reset to older versions, causing a significant loss of code and progress as a result. To mitigate this, we could have utilised Git's branch feature more effectively, creating separate branches for each feature or user story. This would have compartmentalised development and allowed better integration of changes into the main branch, which should have been reserved for final releases alone. However, one positive aspect of this approach was that group members could easily spot problems in each other's recently pushed code as they tested and pushed their own updates. Unfortunately, any potential benefits were outweighed by the issues caused by this approach.

During the envisioning phase, we conducted a risk assessment identifying potential issues that could arise. However, we encountered unexpected issues, such as when we had to generate the JAR file. The issues stemmed from crucial external dependencies not being properly packaged by Maven during compilation. This problem recurred at every deadline, as we could not identify nor resolve the underlying issue. Whilst we ultimately worked around this problem before submission deadlines, it took up time that could've been better allocated elsewhere. We could have mitigated these issues earlier by removing problematic dependencies as soon as they presented an issue, rather than working around them and letting the problem grow with the project. Unfortunately, we identified this too late, when our codebase was already heavily reliant on these dependencies, making changes time-prohibitive. Regularly compiling the JAR after each user story, rather than at the end of an increment, would have helped detect these issues sooner

Looking back, we could have also improved our estimation techniques in relation to our user stories. Initially, we used t-shirt sizing to estimate effort, but this proved to be less accurate than desired. By utilising ideas like planning poker sooner, we could have had more reliable forecasts of work required, allowing for better time allocation. Our communication and collaboration could have been improved, specifically in regards to changing code written by other group members. Although we had regular meetings and used tools like WhatsApp and Microsoft Teams, there were instances where misunderstandings about code-files or lack of clarity regarding changes led to delays or rework. We should have established separate communication protocols for our project's technical aspect, such as those provided by GitHub, to ensure everyone was aligned on changes and progress in the codebase.

All-in-all, whilst these inefficiencies cost us a lot of time, they were learning experiences nonetheless and we hope to better approach these problems in future projects.

2.3 Use of Agile Methodology

2.3.1 Advantages of Agile Methodology

Agile's focus on delivering working software early and frequently provided a sense of tangible progress throughout our journey. First and foremost, by focusing on having a noticeably improved product increment at the end of each sprint, we could demonstrate value to the customer and maintain their enthusiasm in the project. This allowed us to gather user feedback and incorporate it into subsequent increments, ensuring the final product met the users' needs. Adhering to emphasis on continuous improvement through retrospectives proved valuable and we were able to refine our practices as a group and become more efficient over time. This culture of learning and adaptation helped us overcome many obstacles and continuously deliver better results.

The frequent, incremental delivery approach to our GitHub commits allowed us to focus on specific sub-tasks and deliver maximum value with each deliverable, despite the aforementioned issues. This encouraged greater involvement from all group members, allowing everyone to contribute to all aspects the codebase, even those they weren't directly assigned to, and stay updated on all the progress our group was making. Working in smaller increments also made testing easier, simplifying issue detection and resolution during integration.

Treating our customer as a part of the team was a significant advantage for the development process as they were able to stay updated on the application's development thoroughly. This allowed them to approve features or request changes, ensuring the final product met their needs thereby reducing the hypothetical risk of costly changes after delivery. Agile's adaptability allowed us to quickly adjust plans when faced with challenges or changing requirements, for example; due to customer suggestions, or shifting approaches in completing project requirements. Regular reassessment of priorities and progress enabled us to shift focus to where it mattered most, delivering both our promised features and our customer's requests.

Agile also made our group more productive by enforcing self-organisation. It helped us leverage the diverse skills and perspectives of our team members. Another advantage was the transparency and communication with stakeholders. By seeking feedback from our customer in regular review meetings, we ensured that the product aligned with their expectations throughout the development process. This allowed for course corrections and prevented us from deviating too far from the desired outcome. Furthermore, it enabled us to consider new ideas the customer presented; for example, the dark mode feature, airplane runway animation, and an animated user interface.

Were it not for this development philosophy, the customer would have been presented with a generally completed product, but would not have had the opportunity to suggest changes mid-development. Undoubtedly, this also poses risks and challenges which we delve into in the subsequent section of this report, but this approach hugely benefits the customer by making them part of the development process.

2.3.2 Disadvantages of Agile Methodology

Whilst Agile methodologies helped our team, certain aspects presented challenges. Agile relies heavily on customer feedback throughout development. In our instance, this was our supervisor, who was helpful and provided clear guidance. However, when our customer was less accessible, unclear about their needs, requested features that were too time-expensive to implement, development became harder.

Frequent delivery meant that all group members had to put in more time and effort, increasing our workload as we had to deliver a working project every iteration that met all our promised user stories. This was especially challenging when team members had other commitments (exam preparation, other courseworks). Agile also requires a high level of collaboration and communication among team members. Whilst our team worked well together with minimal conflicts, if we had experienced poor team dynamics or interpersonal issues like other groups, it could have affected our progress and work quality. Also, the focus on short-term deliverables can sometimes lead to a lack of long-term planning, something we definitely experienced between increments. Whilst this approach allows for flexibility and modularity, it did result in technical debt accumulation. Balancing short-term responsiveness to deliverables, customer suggestions, and long-term project completion is important, and this is something we should have better anticipated. Agile's face-to-face communication and co-located teams can also present challenges. Team members inevitably missed meetings, but we ensured they were updated through our communication channels to mitigate this. However, had our team been more geographically dispersed, we might have faced far greater hardship in maintaining the high level of interaction and coordination that Agile demands.

Another notable disadvantage was scope creep; with our focus on accommodating changes and incorporating feedback, too much time was spent implementing features beyond our plan. We spent an excess time programming more difficult customer suggestions, causing momentary lapses of focus on our core user stories. It's important to have a clear process for evaluating and incorporating changes, whilst keeping the project goals in mind. It could have proven useful to employ an impact-effort matrix to mitigate this, in which suggestion impact is plotted against effort required, and high-impact, low-effort items are prioritised; thus, reducing scope creep.

2.4 Use of XP Values

Our team utilised principles of Extreme Programming (XP) where possible. To simplify our project, we broke down the complex task of developing the Runway Re-declaration Tool into smaller, manageable segments and focused on the essential requirements given to us. We held regular meetings both online and in-person and used messaging to stay connected, keeping everyone informed about progress, facilitating task delegation, and enabling collaborative problem-solving. Frequently testing our product and discussing it with team members and our customer ensured that our project met customer expectations. A culture of respect allowed us to leverage each team member's strengths while supporting their weaknesses, making it easy for us to be honest about our progress and ask for help when needed. We effectively utilized pair programming, allowing two developers to work together on some user stories, which enhanced knowledge sharing and produced higher-quality code through constructive criticism. By rotating pairs, we ensured that everyone had the opportunity to work with and learn from each other.

The XP principle we adhered to most closely is that of collective code ownership. All group members had the ability to modify and improve any part of the codebase, regardless of code authorship. This policy promoted shared responsibility and encouraged everyone to contribute to the overall quality of the tool. It also simplified addressing issues and making changes quickly, as any team member could step in and work on the code when needed. This was supported by the principle of continuous integration; by frequently merging our code changes into a shared repository and running automated tests, we could swiftly identify and resolve code errors and behavioral issues. This reduced the risk of last-minute surprises at deadlines. By writing tests before implementing complex logic—an example of test-driven development (TDD)—we ensured that our software met the required functionality and behavior. This approach not only aided in developing various parts of the project, such as calculations and importing/exporting, but also encouraged us to think more carefully about the design and requirements of our application, as we had clear objectives for program outputs.

3 Time Expenditure

3.1 Estimation of Time Spent Per Member

The table below describes the estimation of time spent per group member. This estimation is based on our completed sprint plan tables for each increment.

Time Spent (hours)				
Team Member	**Increment 1**	**Increment 2**	**Increment 3**	**Total**
Anthony	11	5	7.5	23.5
Bav	11	6	5.5	22.5
Manlin	10	6.5	6.5	23
Yash	5	6	5	16
Zagrosi	7	5.5	6.5	19

3.2 Most Time Expensive Activities

The project tasks that took the most time were those related to the user interface and implementing different runway visualizations, all core requirements. Our customer also demanded an application that was not only functional but visually appealing, which added significant overhead.

Some features of the runway visualizations, like the auto-rotation, took much longer to implement than initially expected. We had to research and experiment with various approaches to ensure these features worked correctly, which increased the time spent on these tasks. Additionally, a significant amount of time

was devoted to understanding the terminology for airport runways and the calculations required to ensure our application consistently delivered accurate results. This deep dive into domain knowledge was crucial for delivering a reliable product but was time-intensive. The integration of various application components—such as the user interface, runway visualization, and calculation logic—also proved time-consuming. Ensuring all these parts worked seamlessly together often involved extensive debugging and refactoring our code. Testing and quality assurance activities further consumed a considerable amount of time; we had to thoroughly test each feature and the application as a whole to identify and resolve any bugs or performance issues. Writing comprehensive unit tests and conducting manual testing sessions were essential but time-expensive. Lastly, preparing for and attending meetings, including sprint planning and reviews, required significant time investment from all group members. While these meetings were crucial for group coordination and task alignment, they did consume a notable portion of our development time.

In our analysis, we believe that our time expenditure should be assessed with efficiency and direction in mind. Time spent on core project functionality, no matter how lengthy, was not concerning. These tasks were essential to completion, and any extra time spent on them was likely due to a lack of knowledge or expertise in certain technical areas. The most significant source of unproductive time loss was our aforementioned issues with Maven dependencies and compiling our final JAR file (see Areas for Improvement).

3.3 Effort Estimation Method

From the outset of the project, we used t-shirt sizes to estimate the effort or time needed for each user story. We assigned each task a size from S to XL, with S being the least effort and XL the most. We chose this method due to our lack of experience with estimating effort in hours.

As the weeks progressed, we improved at estimating the time required for each task. We also began using story points to provide a more numeric estimation of the relative effort needed for each user story. By assigning points based on complexity and workload, we could better prioritize our tasks and track our progress using velocity metrics. To enhance the accuracy of our estimates, we conducted planning poker sessions during our sprint planning meetings. Each team member independently estimated the effort for a user story, after which we compared estimates and discussed discrepancies. This collaborative approach helped us gain a shared understanding of the requirements and arrive at more reliable estimates. As the project advanced, we started using a combination of historical data and team member experience to refine our estimates. By considering how long similar tasks had taken in previous sprints and accounting for new factors or challenges, we could make more informed predictions about the required effort. We also regularly reviewed our estimates during sprint retrospectives to identify areas where we might have overestimated or underestimated the effort. This continuous feedback loop allowed us to adjust our estimation approach and enhance our accuracy over time.

In addition to estimating development efforts, we also accounted for other activities such as testing, documentation, and deployment when planning our increments. By considering these additional tasks, we could create more realistic and comprehensive estimates that better reflected the total work required to deliver each user story.

3.4 Workload Balance

Early in the project, we discussed each team member's strengths and weaknesses. Some of us were better at creating user interfaces, while others were more skilled at implementing features like input validation, calculating runway parameters, and testing. By assigning tasks based on each person's strengths and being aware of their weaknesses, we could support each other and distribute the work more effectively. This approach helped us make the most of our team's collective skills and ensured that the project progressed smoothly.

We also made sure to regularly check in with each team member during our meetings to see how they were coping with their assigned tasks. If someone was struggling or had too much on their plate, we would redistribute the work or provide additional support to maintain a balanced workload. When faced with particularly complex or time-consuming tasks, we used pair programming to tackle them together. By having two team members work on the same task simultaneously, we could share the cognitive load, catch errors more quickly, and ensure that no one was overwhelmed by a challenging piece of work. In addition to technical tasks, we also rotated responsibilities for project management, documentation, and other non-development activities. This ensured that everyone had a chance to contribute to different aspects of the project and prevented any one person from being overburdened with administrative duties. To further balance the workload, we made use of Git to visually track the distribution of tasks across team members. This allowed us to easily identify any imbalances and make adjustments as needed.

By continuously monitoring and adapting our workload distribution throughout the project, we were able to maintain a healthy balance that kept everyone engaged and productive without causing burnout or undue stress.

4 Tools and Communication

4.1 Software Engineering Tools

Name	Description	Evaluation	Effectiveness
IntelliJ	Integrated Development Environment	Familiar to group members due to past modules. Useful extensions utilised (e.g. GitHub, allowing codebase collaboration).	4/5
JUnit	Unit testing framework for Java projects	Helpful for testing functionality of the application, e.g. calculations, import and export XML functions, input validation.	5/5
Maven	Dependency framework for Java projects	Caused severe issues with dependency installation, and project compilation. Very difficult to learn to utilise in little time.	1/5

4.2 Collaboration Tools

Name	Description	Evaluation	Effectiveness
GitHub	Version control framework	Used for version control, code collaboration between team members, not utilised to full potential.	4/5

4.3 Communication Tools

Name	Description	Evaluation	Effectiveness
Microsoft Teams	Platform used for group meetings and assessments	Used to host multiple online meetings, and for team members who couldn't attend in-person meetings. Familiar to group members due to university-wide usage, ensuring effective use of helpful features for group communication.	5/5
WhatsApp	Instant messaging platform, used for quick group communications	Efficient & accessible for team member queries regarding code, documentation, or past/upcoming meetings. Also used to notify team members on sub-task completion, compensated under-utilisation of GitHub commit notes.	4/5

4.4 Meeting Strategy

Type	Description	Weekly Freq.
Physical	The team would hold in person meetings via booking a room in the library for 1-2 hours to review the current state of of the document for a deliverable as well as the current state of the code. The review for both the code and document would take 20-30 minutes of the meeting. For the remainder of the meeting we'd perform sprint planning and than depending on the tasks at hand and everyone's availability, either work on tasks in the meeting and work on the tasks at home	1-2
Online	The online meetings would be scheduled via Microsoft teams and would generally last around 30 minutes to an hour. During the online meetings the team would review both the code and the current deliverable document as well as convey any queries that concerns each individual's tasks for the deliverable. In some of the online meetings, if there was a heavily loaded user story or an issue concerning the document, the online meeting would become a work meeting whereby as a team we tackle how to implement a code for the user story or how to resolve an issue in the deliverable document.	1-2
Supervisor	During the supervisor meeting, our supervisor would go through the specification for the current deliverable and give us tips on how to perform aspects required from the specification. After the supervisor briefs the team on the current deliverable specification, the team will show the current application and ask questions regarding both the code and document in order to better improve the project as a whole.	1
SCRUM	A short meeting which generally lasted around 10-15 minutes whereby the team would elect a scrum master who discussed with every team member, the developer team, what work needs to be done and gives feedback on the current state of the code and deliverable document. For the start of this project, the team employed daily SCRUM meetings to ensure everyone was well notified on what work needed to be done as well as reviewing the current progress. As the project neared completion, we reduced the quantity of SCRUM meeting per week as we had completed the majority of major tasks during the start of the project.	3-7

5 Advice to Next Year's Students

Throughout our project, we faced various challenges, some expected and others unexpected. Based on our experience, we would like to offer the following advice to students taking on this project next year:

- Read the specifications for both projects thoroughly before making a choice. One project will likely align better with your group's strengths. Choosing the right project will make your work much easier.
- Familiarise yourself with the specifications of your chosen project before starting work. This will result in fewer changes needed in later iterations and allow for better planning.
- Have frequent meetings with group members and regular contact with your supervisor. This will help you stick to timelines and ensure the project stays on track.

- Implementing most features in the initial increments has benefits, but doing too much work in a short time can lead to burnout. Plan appropriately, considering the needs of all group members.
- Adopt agile methodologies from the start. Embracing practices like Scrum, XP, and continuous integration will improve collaboration, adaptability, and code quality. Don't underestimate the value of regular meetings, retrospectives, and pair programming.
- Prioritise effective communication. Establish clear channels for sharing ideas, progress updates, and problem-solving. Use tools like messaging apps and project management software to keep everyone informed and on the same page.
- Pay attention to version control and code organisation. Use Git and branching strategies to manage your codebase effectively. Agree on coding conventions and stick to them consistently to maintain code readability and maintainability.
- Don't neglect testing and quality assurance. Allocate time for writing unit tests, conducting manual testing, and performing code reviews. Catching and fixing issues early will save you time and effort in the long run.
- Be proactive in identifying and mitigating risks. Conduct a thorough risk assessment during the envisioning phase and create contingency plans for potential technical and non-technical challenges.
- Remember to document your work. Maintain clear and concise documentation, including user stories, design decisions, and project progress. This will help you stay organised and make it easier for others to understand and contribute to your project.
- Foster a supportive and collaborative team environment. Encourage open communication, respect each other's ideas and contributions, and be willing to help one another when needed. A strong team dynamic is crucial.

6 Conclusion

In conclusion, our team's journey through the Software Engineering Group Project has been a valuable learning experience. We successfully developed a Runway Re-declaration Tool that met and exceeded requirements. This project has provided us with hands-on experience in applying agile methodologies, collaborating effectively, and delivering a successful software product. The challenges we encountered and the lessons we learned have enriched our understanding of software engineering principles and prepared us for future large-scale collaborative projects.