Quality Assurance Manual

First Edition

Edited by
Daniel Bishop
Oliver Clarke
James Gardner
Stijn Marynissen
Che McKirgan
Cameron Smith
Oliver Still
Eric Walker

Group 2
MEng Software Engineering Project
Department of Electronic Engineering
University of York

2019 / 2020

Document Version Control

History of edits and alterations to the CUBIXEL quality assurance manual, including the document version, date, author, and description of the edits.

Version numbering is based on the significance of change.

Key: (major-change/milestone**.**new-section**.**section-edit)

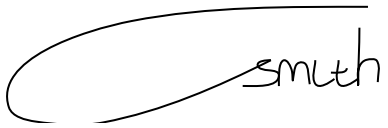
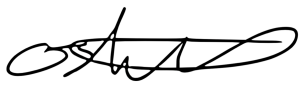
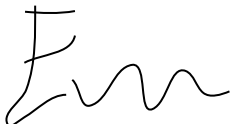
Version	Author	Date	Section Modified	Remarks
0.1.0	JG	13/11/19	All Sections	Document created and basic structure laid out.
0.1.1	OC	20/11/19	2. Roles and Responsibilities	Editing finance role section.
0.1.2	JG	21/11/19	2. Roles and Responsibilities	Editing technical manager section.
0.1.3	SM	24/11/19	2. Roles and Responsibilities	Editing design manager section.
0.1.4	EW	25/11/19	2. Roles and Responsibilities	Editing Testing and Integration Manager role section.
0.1.5	JG	25/11/19	2. Roles and Responsibilities	Editing Quality Assurance Manager Section.
0.1.6	CS	25/11/19	2. Roles and Responsibilities	Editing Marketing Manager section.
0.2.0	JG	27/11/19	4. Project Management Methodology	New section on Project Management Methodology.
0.2.1	OS	28/11/19	1. Introduction 2.1 Organisational Structure	Edited Introduction with the company's mission and vision. Included diagram and description of company hierarchy.
0.2.2	OC	29/11/19	4.4.2 Risk Management	Filled section in and should be complete.
0.2.3	JG	29/11/19	Preface 4.4.3 Continuous Integration 2. Roles and Responsibilities 5. Facilities and Safety 6. Equipment	Preface/Abstract written. Main description of User Stories completed. Multiple team role sections updated. Other Sections.
0.2.4	OS	29/11/19	1. Introduction 2.3 Project Manager 4.4.1 Task Organisation	Finished drafting introduction. Finished drafting Project Manager section. Started Task Organisation.
0.2.5	CM	29/11/19	10. Information Management 2.6 Technical Manager	Edited information management section. Edited Technical Managers risks.
0.2.6	OS	1/12/19	4.4.3 User Stories	Edited methodology for calculating prioritisation and velocity.
0.2.7	EW	01/12/19	4.5.3 Exploratory Testing	Started draft.

0.2.8	CS	01/12/19	4.2.1 Marketing 4.1.3 Root Cause Analysis 4.2.2 Stand-Up Meetings	Edited Ideals section. Edited Collaborating section.
0.2.9	JG	02/12/19	4. Project Management Methodology	Created Marketing and Design Sub-Sections and added some draft flow charts for the overall project timeline and iteration timing, these need more work. Reorganised Section 4 into more logical order.
0.2.10	EW	04/12/19	2.9 Testing and Integration Manager 4.2.1 Customer Involvement 4.4.2 No Bugs 9.3 Eliminating Waste	Added to the Testing and Integration Manager role section. Drafted Customer Involvement, No Bugs sections, Eliminating Waste and Seeking Technical Excellence.
0.3.0	CM	04/12/19	4.4.4 Collective Ownership 4.3.2 Refactoring 8.8 Non-Permanent employees. 9.2.4 Employee satisfaction.	Created Collective Ownership section. Created Refactoring section. Created Non-Permanent Employee section. Created Employee satisfaction section.
1.0.0	**All Members**	**04/12/19**	**All Sections**	**Initial Draft Complete**
1.0.2	OS, JG	14/12/19	All Sections	Formatting.
1.0.3	CS	16/01/20	Cover Page 2.5 Marketing Manager	Re-Designed cover page and made edits to the Marketing Manager section.
1.0.4	EW	18/01/20	2.9 Testing and Integration Manager	Added QA metrics.
1.0.5	JG	21/01/20	2.6 Technical Manager 2.8 Quality Assurance Manager	QA metrics updated and improved.
1.0.6	OS	21/01/20	2. Roles and Responsibilities	Rephrased risks and QA metrics to follow the same format.
1.0.7	OS	22/01/20	4.1.2 Task Organisation	Added information about the notice board and updated text.
2.0.0	**All Members**	**30/01/20**	**All Sections**	**Document approved by team.**
2.0.1	JG	02/02/20	4.1.7 Git Commits & the Iteration Cycle	Updated iteration duration for first iteration.
2.0.2	OS	03/02/20	2.3 Project Manager	Added info about managing the Trello boards and updated Risks.

2.0.3	OS	12/02/20	2. Roles and Responsibilities	Rephrased all QA metrics to include more measurable metrics with time frames.
2.0.4	JG	24/02/20	2.6.2 Technical Manager Risks	Added more risks and solutions to technical manager.
2.0.5	OC	27/02/20	2.4.3 QA Metrics	Changed finance metric.
2.0.6	OS	27/02/20	4.1.5 User Stories	Removed all references of velocities and swaps for time estimates or ideal days.
2.0.7	OC	05/03/20	2.4.3 QA Metrics	Changed two other metrics.
2.0.8	JG	09/03/20	2.8.2 QA Manager Risks 4.2.3 Project Structure Updated.	Added more QA Risks and included a new image of project module file structure. Updated version numbering to match other documents.
2.0.9	JG	11/03/20	2.8.2 QA Manager Risks 4.2.3 Logging Updated.	Added risk to QA Manager and created a default XML for Log4j.
3.0.0	**OS**	**12/03/20**	**4.1 Planning** **4.4 Releasing**	**Updated to match real methodology.**
3.1.0	OS	25/03/20	10. Disaster Recovery	Added a new section about the company's disaster recovery and working from home.
3.1.1	JG	10/04/20	4.2.3 Logging	Improved the logging section to match current practice.
3.1.2	OS	12/04/20	10.1 Risks	Small edits to risks.
3.1.3	CS	18/05/20	4.7 Marketing	Progressed marketing section.
3.1.4	OS	20/05/20	2.3 Project Manager	Reworded QA metrics.

Document Approval

All authors of the document are required to proofread, mandate, and sign-off before the document's official publication.

Author	Signature	Date
Daniel Bishop		12/03/2020
Oliver Clarke		12/03/2020
James Gardner		12/03/2020
Stijn Marynissen		12/03/2020
Che McKirgan		12/03/2020
Cameron Smith		12/03/2020
Oliver Still		12/03/2020
Eric Walker		12/03/2020

Preface

This document is the Quality Assurance Manual for the company CUBIXEL. It provides the reader with an understanding of what CUBIXEL values and represents. It details the corporate structure of the company and information about the responsibilities of each role. The document serves as a reference for how the company's software engineering projects should run, providing details on many aspects of CUBIXEL's particular approach to project management and is updated regularly to reflect the dynamic and continually improving nature of the company.

The company, Cubixel ("we", "our", "us"), may be stylised as "CUBIXEL" and may be referred to as the COMPANY.

Contents

1. Introduction

1.1 Company Overview

CUBIXEL is an eight-person software engineering consultancy and development team based in York, United Kingdom. As a team, we provide software design services to clients and large companies, focusing on full stack development and providing true customer value using agile development practices.

We are inclusive to all industries that align with our own company ideals, especially those involved in global change to provide a positive impact to fields of research, and in people's homes.

1.2 Mission Statement

At CUBIXEL, we strive to provide companies with the necessary tools to prosper in their respective fields. Our business focus is on the design, implementation, and support of research and academic based software for companies willing to push the boundaries of education and research. Our mission is to lead the way into providing effective and accessible educational services through advanced and innovative means, capitalising on new business opportunities and delivering industry-leading technology.

With the core of our team initially meeting at university, we understand the importance of the social tools required to aid those in higher education. By contributing our knowledge and expertise to the academic field of study, we believe we can establish and succeed in new areas of research and advance society to educate more young people into STEM and social based sectors.

1.3 Vision Statement

Our vision for the company is to advance new areas of education and academia through the development of software to encourage more students into STEM based subjects. CUBIXEL aims to be one of the leading developers of software for enabling education, research and collaboration. CUBIXEL plans to expand quickly whilst maintaining its core values and a strong sense of pride in the projects we develop amongst all employees.

2. Roles and Responsibilities

2.1 Organisational Structure

The structure of the company is designed to spread responsibilities while also enabling collaboration in order to prosper participation and teamwork. Select managerial roles, typically higher within the company, are undertaken by a sole individual, whereas more general and cooperative roles are managed by multiple members of the team. This organisation eliminates any discordance or mismanagement in higher positions that would otherwise lead to disorganisation and a possible lack of clarity, while also providing the members in these roles the control, responsibility, and focus to direct the team's development. Figure 1 shows a diagram representing the structure and hierarchy within the company:

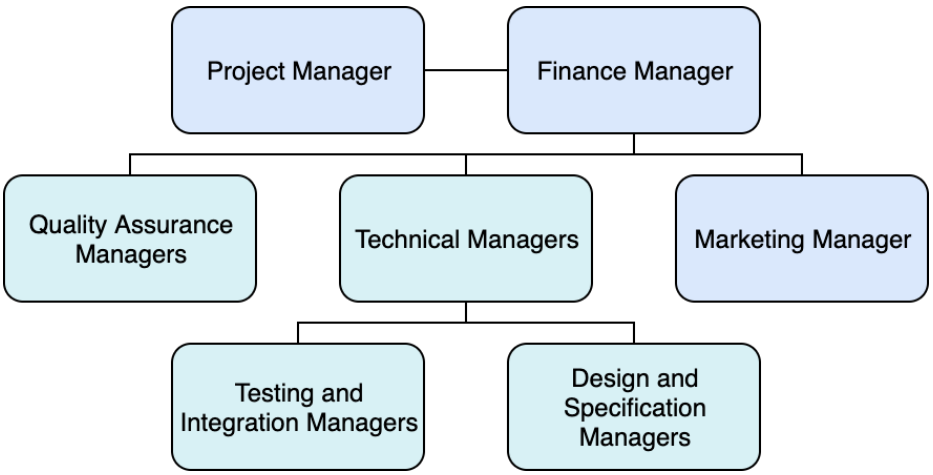


Figure 1 - Diagram representing the company hierarchy. The project manager, finance manager, and marketing manager positions are undertaken by a sole individual (in dark blue).

2.2 Whole Team

The CUBIXEL team sits together in an open workspace. The company works on an iteration basis and each iteration should start with the whole team meeting in person, showing an iteration demo to the customer, a retrospective discussing how things went that iteration and planning the next iteration.

All members of CUBIXEL are expected to be independent and plan their own work, no explicit schedule for any task will be given, other than iteration timing. It is the responsibility of all members to prioritise how they will approach a task; this should be done with their pair programming partner for the respective task.

Meetings can and should be organic; discussing all issues and ideas in person is encouraged. This will make resolving issues and customer requirements quick and easy. The entire team is responsible for coding standards, and it is expected that if someone spots an issue with any code, then that person should fix it as soon as possible.

All members of the team should track the hours they work on any task for CUBIXEL on Clockify during or immediately after completing that task.

Everyone should read the QA Manual and understand their responsibilities and their associated risks within the company prior to starting their respective role and after any major updates or new sections are added to the document. CUBIXEL is always looking to improve and any new ideas are encouraged.

2.3 Project Manager

2.3.1 Role Description

The project manager is responsible for planning, organising, and leading the project's development team and curation process. Additionally, they help manage communication between the team and the rest of the company to ensure clarity and a unified direction. Their goals are to achieve a high level of performance and quality to their relative client and its customers, as well as to achieve personal, technical, and organisational successes within the company.

Key Responsibilities:
On a business level, requirements include:

- Managing a well-defined project management process leading large, complex enterprise-level projects.
- Setting achievable goals and deadlines within the project's development, managing stakeholder communications, and supporting the product owner to ensure realistic customer expectations for project deliverables are maintained.
- Deliver projects with exceptional value to the respective client within the designated project constraints.

At a team level, requirements include:

- Assigning the appropriate people, processes, and tools, to improve team efficiency and effectiveness during project development.
- Ensure that each team member is fully engaged in the project and making a meaningful contribution and encourage a sustainable pace with high levels of quality for the team.
- Assist in team development while holding team members accountable for their commitments, removing roadblocks to their work, and leveraging company resources to improve capacity for project work.
- Curating and managing the team Trello boards to keep the team updated of upcoming meetings and to track project developments (see section 4.1.2)

2.3.2 Risk Management

Risk	Possible Solution
Mismanagement of project or team from project manager.	Weekly meetings with the core members of the team. Regular communication and updates with all members of the team to ensure clarity and a unified direction.
Missed agreed project iteration deadline.	Realistic deadlines initially set. Allow team members to contact about their schedule. Regular updates with the client about the progress and scale of the work.
Project iteration does not meet the client's standards.	Meetings setup with clients to ensure projects are delivered to their standards. Any internal changes or updates are communicated directly to the client.
Low employee morale or employee burnout.	Allow team members to book days off in advance. Regular work days out organised outside of the office space. Sequentially check in on the team and their working hours to check their work rate for any signs of burnout.

Loss of a team member.	Label all possible consequences of the team member's leave and reallocate tasks to other employees based on their knowledge and ability. Reschedule project deadlines and hold meetings to inform the team and client on those changes.
Inconsistency of data between third-party management applications.	Regularly check that data is consistent across all company management pages. Have one central application (e.g. Trello) to always contain up-to-date information on the company. Additionally make sure all employees are aware of changes verbally.

2.3.3 QA Metrics

Metric	Measurement
Work Efficiency	Mean average number of hours to complete an assigned user story at the end of a single project iteration.
Team Productivity	Average working hours per day of the whole software development team at the end of a week every Thursday.
Team Contentment	Number of overall positive responses in individual feedback from each team member on the management and organisation of the project before each iteration.
Reachable Project Scope	The overall percentage iteration completion of the assigned user stories at the end of the iteration period.
Client Satisfaction	Percentage of number of positive responses against negative responses from the client / on site customer after each project iterations/revisions during the project's lifespan.

2.4 Finance Manager

2.4.1 Role Description

The finance manager records, documents and budgets the project expenditures and timesheets, as well as assisting the project manager by closely communicating the project's business development. They are also responsible and authorised to accept and suggest changes to the project's requirements, budget, and financial timelines by members of the team and the on-site customer.

Key Responsibilities:

In the financial sector of the business, the finance manager is responsible for:

- Collating, preparing and interpreting reports, budgets, accounts, and financial statements.
- Monitoring and interpreting cash flows and predicting future trends.
- Controlling income, cash flow and expenditure for the business through the company's financial accounting.
- Producing accurate financial reports to specific deadlines.

As a member of the team, the finance manager is responsible for:

- Formulating strategic and long-term business plans.
- Advising the project manager.
- Authorising changes submitted by members of the team to the project's requirements, budget, and deadlines.
- Develop professional external relationships with appropriate contacts (e.g. bankers).
- Liaising with managerial staff and other colleagues.
- Tracking project progress with respect to budgets.

2.4.2 Risk Management

Risk	Possible Solution
Accumulating amounts of monies owed which are not affordable.	Make detailed and sensible contracts with customers and investors. Ensure initial predictions of the project scope/cost are realistic from the start with contingency.
Funding pulled from financial backers.	Set realistic project expectations between the company and the financial backer. Perform a background check on the financial backer to avoid any financial or fraudulent misconduct.
Company running out of budget before the product's completion.	Monitor budget regularly compared to current project progress and adjust when necessary. Check that budget matches with predictions of product deadlines and ensure that this time remains within budget.
Borrowing an excessive sum of money from financial backers.	Return excess borrowed money to financial backers within reasonable time to avoid unnecessary financial backing and interest paid on loans.

2.4.3 QA Metrics

Metric	Measurement
Percentage Iteration Progress Expenditure	Difference between percentages of expenditure and completion for each week every Thursday.
Labour Buffer	Remaining surplus hours from the budget which can be dynamically assigned to employees for overtime. Tracked weekly every Thursday.
Development Cost Difference	Difference between budgeted programming cost against actual iteration cost. Tracked fortnightly every Thursday.

2.5 Marketing Manager / On-Site Customer

2.5.1 Role Description

The Marketing Manager is crucial to ensuring that a company is successful in promoting and selling its product to consumers; influencing the way a brand and/ or product is perceived in the market.

Key Responsibilities:

- Develops all global strategic and tactical marketing initiatives and plans, in line with company objectives.
- Leads the implementation of the company marketing strategy - including campaigns, events, digital marketing, and public relations.
- Undertakes analysis of competitive environment and consumer trends.
- Manages social media presence and directs programs to improve social media reputation and recognition.
- Maintains effective internal communications to ensure that all relevant company functions are informed of marketing objectives.

On-Site Customer

An On-site Customer is essential to increasing the value of the product that the company delivers; the product must also bring value to its investors; this requires the perspective of on-site customers.

Key Responsibilities:

- Determines what stakeholders find valuable; defining the project vision and therefore the software that the team builds.
- Identifies features and stories that would increase the value of the software to its investors.
- Constructs and refines project release plans by soliciting feedback from stakeholders and coordinating with programmers.
- Provides programmers with details of requirements and communicates by producing mock-ups, reviewing work in progress, and creating detailed customer tests that clarify complex business rules.

2.5.2 Risk Management

Risk	Possible Solution
Product competition after it is released.	Ensure quality of product throughout the project. Provide a differentiated customer experience by offering reliable and accessible customer support and by ensuring product maintenance of service.
Marketing strategy is unsuccessful or inappropriate to the product's field or image.	Seek/employ temporary professional advice if within budget. Allow additional experienced staff to the marketing team. Involve client in the marketing methodology to guarantee expectations are met.
Insufficient secondary market research data.	Conduct primary market research using methods such as general online surveys and focus groups to collate data. Check data samples are sufficient to validate results.
Low average product rating from client.	Conduct secondary research to determine the reasons for the low average product rating, this can be done by offering the client the opportunity to justify their rating. Offer customer support product maintenance services to derive a solution in order to rectify the issues raised by the client.

2.5.3 QA Metrics

Metric	Measurement
Product Awareness - Profile Visits	The total number of people that have visited/ viewed the product social media account over the period of a week.
Product Awareness - Profile Reach	The total number of people that have viewed/ interacted with any of the posts made by the product social media account over the period of a week.
Product Growth	The total number of people following the product social media account at the time of measurement.
Product Demand	Percentage of surveyed people currently using or looking for the product or similar products, measured at the beginning and end of the development period.

2.6 Technical Manager

2.6.1 Role Description

The Technical Manager is the lead programmer within the development team. They are responsible for the underlying architecture of the software being developed and they oversee and sign-off all work done by junior developers. The Technical Manager should act as a mentor to new members of the development team encouraging an Agile mindset in new employees and helping them to quickly integrate into the company. They are to maintain and promote the use of Agile development principles amongst all software developers working alongside QA to provide measurable metrics for the company's Agility.

Key Responsibilities:

- Sets coding standards and ensures all software follows company style guidelines.
    - Maintain documentation on a per language basis for coding style guidelines.
    - Ensure all software developers know where to find style documentation.
    - Check all code for style before being committed to the master branch.
    - Lead discussions and have final say on development tools used.
    - Lead discussions and have final say on file and directory layout.
    - Lead discussions and have final say on build conventions.
    - Lead discussions and have final say on logging conventions.
    - Lead discussions and have final say on design conventions.
- Champions the use of Agile development processes and work alongside the QA Manager to measure company Agility.
    - Ensure all software design is following a Test-Driven Development approach.
    - Encourage in-person discussions over software issues.
- Supporting software engineers to follow architectural best practices.
- Escalate product opportunities to the team and company director.
- Resolves technical disputes.
- Lead the research of new technologies.
- Relates technical language to clients in an understandable way.
- Manages and maintains software version control and the git repository.
- Producing invoices for software development contracts.
- Assists the Finance Manager in curating the project timeline by assessing 'User Story' difficulty alongside the software developers.
- Assist the Testing and Integration Managers with the creation of the Test and Integration Plan. Aid in some testing performed.

2.6.2 Risk Management

Risk	Possible Solution
Large amounts of time required to learn new technology.	Giving a definitive time limit for each Spike into new technology and then assessing the viability of implementing that technology at the end of the Spike with the programmer involved.
Misinterpretation of code due to inexperience or absence.	Rotate the pairing partners so programmers work on multiple sections and with each member of the software development team.
Disagreement/Non-compliance over technology and approach within the software development team.	Both parties will present a case for their technology/approach and the final decision will be made by the Technical Manager.

Disagreement over project scope between programmers and management.	All timings are to be agreed upon by the project manager and the technical manager with input from the developer assigned to the task.
Team members not following TTD practices.	Speak to team members to understand why they aren't following the process. Understand issues preventing this and work together to find a solution.
CUBIXEL is not remaining up to date with new technologies.	Give time for team members to research new technologies. Have the team report back on the success of research and possible applications within the company.
Client expectations misaligned with reality.	Speak to clients often and ensure they understand limitations and possible solutions. Give honest answers to clients demands.

2.6.3 QA Metrics

Metric	Measurement
Software Development Team Agility	Using the 'Assess Your Agility' self-assessment quiz. This should be done at the beginning of each iteration to review the previous iterations performance. Lowest Value Recorded.
Time Estimates for User Stories	Difference between estimated time and actual time for completion of user stories so as to refine estimates. This should be done at the end of each iteration as a sum of all stories. Further time estimates should then be updated based on the insight from this check.
Non-Compliance for Coding Standards	A code review should be done for each story completed and marked as 'checked' alongside the user story card. Any non-compliances should be highlighted and recorded. The number of non-compliances should be reduced as the team learns the coding standards. Code should also be reviewed for adherence to standards imposed by outside regulatory bodies as and when required.

2.7 Design and Specification Manager / Interaction Designer

2.7.1 Role Description

The Design and Specification Manager holds two major responsibilities within the team: creating and refining user experience, therefore being one of the driving forces in front end development along with the Marketing Manager; and scrutinising each iteration before release, making sure that the iteration is compliant with the specification. If the Marketing Manager can be defined, in part, as an on-site customer; the Design and Specification Manager can be represented as an on-site user. Under the Agile development philosophy, the design (then consequently implementation) of the program is dynamic and subject to change with each iteration as beta users and customers react and critique to the new version of the application.

Key Responsibilities:

- Designing and implementing the GUI and user control.
    - Researching general and project-specific user behaviour.
    - Consideration of defied project vision and how that is realised.
    - Designing user interaction (how the user will mechanically use the application i.e. the "flow" of the program).
    - Designing aesthetics (how the program looks to the user) i.e. icon design, interactable component placement, font typing and diversity, colour pallet, use of positive and negative space, etc.
- Assisting the Lead Developer in designing the software code model.
    - Communicating the mechanics of interaction models and aesthetic choices made so they can be implemented by the back-end division of the team.
    - Discussing programming realities with the back-end team so models and choices can be reworked/reconsidered, if different approaches are necessary.
- Checking the software matches the specification / brief (quality control).
- Overseeing playtesting of software and interviewing users on appropriate iterations (every time a big enough front end change in the program occurs).
- Documenting design choices and developments.
- Presenting and explaining design choices/developments to the whole team, with possible team input and discussion.

2.7.2 Risk Management

Risk	Possible Solution
Uncomfortable user interaction with the program.	Research user's wants and needs, playtest programs with external beta users and incorporate user feedback in the next iteration.
Project vision not sufficiently realised.	Clear communication with the Marketing Manager (acquiring customer wants and needs) and the Technical Manager (communicating design, based on a researched model of user interaction derived from and limited by the criteria acquired from the Marketing Manager).
Released iteration is not to specification.	Collaboration with the Testing and Integration Manager (making sure testing procedures are thorough enough) and implementing strict peer reviewed quality control.

Client's design proposals are too costly to be worth implementation.	Frank communication with the client about programming realities of certain desired designs.
Playtesting software leads to software leaks.	Making playtesters sign a Non-Disclosure Agreement to legally protect information about the software from data leaks. Also using feedback from team-members playtesting the software as to reduce the amount of outside exposure of the software.
Falling behind on design documentation.	Sticking to a policy of producing documentation or a sketch of documentation immediately after making design decisions or reworks.
Failure to accurately present design developments to the team.	Quick but comprehensive follow ups with individual members of the team to make sure they understand the design direction of the program.

2.7.3 QA Metrics

Metric	Measurement
User Experience Satisfaction	Difference in number of positive responses against negative responses gained from focus groups and user interviews on using the product directly. This is measured at the end of each iteration, starting from the 2nd iteration.
Similarity to Design Vision	Difference in number of positive responses against negative responses from feedback and discussions with the client on the product design philosophy and specification. This is measured at the start of every Thursday, starting from the 2nd iteration.
Complementation to Project Vision	Number of redesign iterations initialised by the client or on site customer, including removed design features. This is measured continually throughout iterations with a total number of redesigns for each work week every Thursday.

2.8 Quality Assurance Manager

2.8.1 Role Description

The Quality Manager works with all members of the company from all levels to set and review the standards of the company. They then ensure the company is meeting its own standards through the use of statistical reviews and audits. They aid all managers in producing the documentation for their Quality Assurance Metrics and help provide insight and solutions to information obtained from those metrics. The QA Manager is also responsible for ensuring every member of the company is aware of the company's vision, standards and ideals.

Key Responsibilities:

- Produce and maintain the Quality Assurance Manual.
    - Take employee feedback on company policy regularly to ensure the company is continuing to develop.
    - Continually review processes with all company managers to ensure current project management methodology is the best fit for CUBIXEL.
- Documenting all team meeting's minutes and make sure all other meetings also use the minutes template.
- Communicating company policy changes and information to all employees.
- Assists the Finance Manager in documenting technical and business requirements.
    - Producing documents to the required standards.
    - Ensuring all documents use the company layout and colour scheme.
- Devise procedures and documents to inspect and report quality issues.
- Facilitate proactive solutions by collecting and analysing quality data from audits and assessments.
- Review current standards and policies within the company.
- Keep records of quality reports, statistical reviews and relevant documentation.
- Ensure all legal standards are met.

2.8.2 Risk Management

Risk	Possible Solution
Company not performing to standards set in QA Manual.	Highlight areas the company is not meeting standards set to all members of the team and discuss possible root causes of the issue to find solutions and prevent reoccurrence.
Software development team not working to Agile standards.	Use the Assess Your Agility test document to identify the areas that need improvement and decide as a team how to address those issues. Review progress over the next iteration.
Changes in company policy not known by all members of the company.	Email all staff regarding any changes to company policy, with employee signatures if required. Note any employee feedback on policy changes.
Data collected on company processes is not useful.	Check with managers often that the QA Metrics are providing value and change them if not.
Legal standards throughout the company not being met.	Conduct audits regularly of company processes and products to check for compliance to legal standards. Seek out advice from legal experts if non-compliances are found.

2.8.3 QA Metrics

Metric	Measurement
Coherence to Management QA Metrics	Use the assess the 'QA Metric Testing' document and check that all QA Metrics have a test and are being satisfied. This should be done at the end of every iteration, measured as a percentage of metrics being currently assessed.
Employee Comprehension of Company Ethos	Complete random audits once every four weeks on one member of the team using the 'Team Checker' document. Number of non-compliances noted and raised if necessary.
Document Standardisation	Company documents proofread every other iteration with any issues noted and raised. Number of non-compliances recorded and highlighted on the 'Document Checker' document.

2.9 Testing and Integration Manager

2.9.1 Role Description

The Testing and Integration Manager is responsible for ensuring that the software is tested against the project specification, as well as overseeing the integration phase of development to prevent faulty code from being merged into the main branch by performing exploratory testing. The results of this exploratory testing, if bugs are found, should be used to help ensure that programmers produce better code in the future by using root-cause analysis to discover why bugs are being produced. If the code is found to be faultless, the tester is then responsible for passing it down the chain to be marked as 'Done Done' (as described in 4.4.1) and integrating the new code with the main branch when the story is 'Done Done'. They are also responsible for assisting programmers in producing module tests for TDD, as they will have an overarching view of how the software will work, and so how it should be tested.

Key Responsibilities:

- Ensure that programmers are producing code with minimal bugs.
- Assist in producing automated tests for TDD.
- Handling integration of new code to the main branch.
    - Ensuring that new code is as bug-free as possible. If it is not, discovering why, and fixing the problem, both in the code and in the methodology that produced the code.
    - Ensuring that integrated code is the correct version.
- Test complete software for functionality and to find bugs.

2.9.2 Risk Management

Risk	Possible Solution
Faulty code being merged with master branch.	Exploratory testing to ensure the code being produced by programmers has no noticeable bugs, and to catch any bugs that are produced.
Produced code containing significant bugs.	Root-cause analysis to discover why bugs were produced and modify programmer's methodology to stop recurrences.
Programmers commit code to master branch without necessary checks.	All code must pass an approval process detailed in section 4.4.1.

2.9.3 QA Metrics

Metric	Measurement
Quality of Code	Number of bugs found in code in a working week via exploratory testing of the simulated program before code is subject to testing.
Quality of Tests	Expected test outcome against the actual test outcome, record the proportion that arises due to insufficient module tests via exploratory testing for a working week.
User Story Integration Latency	Measure and record the length of time between each user story being submitted for 'done' exploratory testing and being integrated into the development repository each iteration.

3. Deliverables

Deliverable	Notes
First Customer Meeting	Initial meeting with customers to discuss the viability of the project. This should involve the project manager, financial manager, technical manager and marketing manager.
Functional Specification	Agreement between the company and the client on the project being undertaken. This is developed alongside the customer and defines the roles and responsibilities of all parties involved.
Project Financial Report	Estimate of project costs and future development/maintenance cost estimates. An initial report of project cost projections is delivered alongside the function specification to the client. Regular financial reports of the project are delivered to the client with each iteration.
Fortnightly Iterations	Functioning build at the end of each iteration, further details about project iterations are provided in section 4. The client demo is described in section 4.5.1.
Final Customer Demo	Final demo confirming all customer requirements defined in the original functional specification have been met.
Project Documentation	Product documentation detailing functionality of the software, development challenges and solutions used with reasons provided. This should also include a user manual.

4. Project Management Methodology

4.1 Planning

4.1.1 Project Level Flow

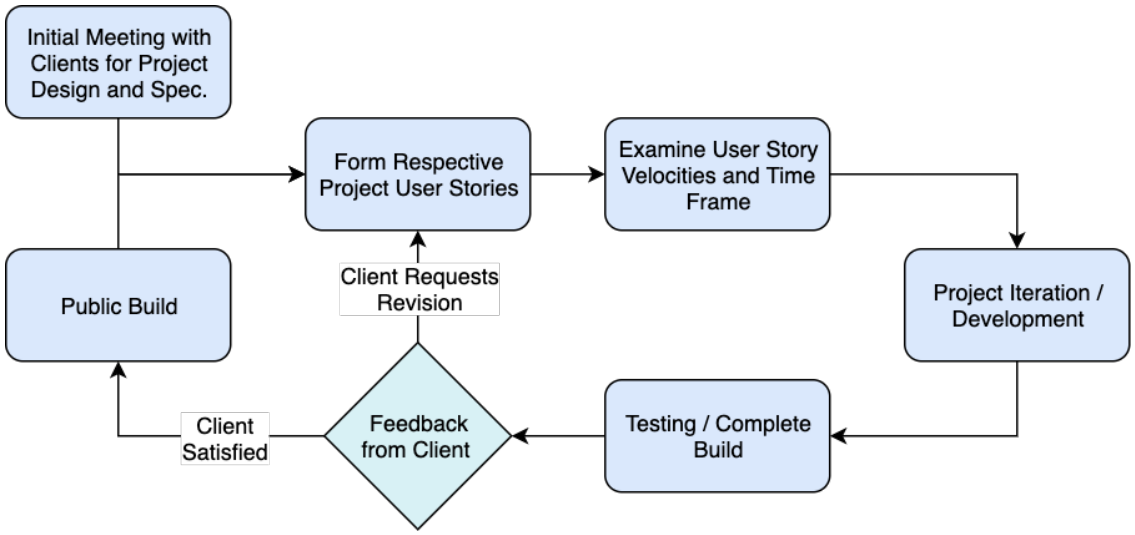


Figure 2 - Overall project flow chart for general project management process.

4.1.2 Task Organisation

Project tasks are organised within the online service Trello and are overseen by the project manager. A personal work account is given to all employees, as well as being added to two workspaces, to allow Trello to act as the central online hub of the company and to aid in project management.

The 'Notice Board' workspace contains all company resources and administrative tasks, including important documents and manuals, and any upcoming/past meetings. The 'Project Development' workspace organises all tasks relevant to the current company project. This workspace has been automated to sort the tasks based on their current state and timeframe within the project.

Individual tasks are first manually assigned a feature ID, user story, tests, due date, label, assigned team members, time estimates, notes, and a GitHub project development branch before coding can start. All team members may move these tasks as necessary between 'To Start', 'In Development', 'In Testing', and 'On Hold' lists.

Once all tests have been completed on the task's card, the card is automatically moved to the 'Done' list where it will then be reviewed by the Project Manager. Once the approval criteria are additionally complete, the card is then moved to the 'Done Done' list where it can then be committed to the development project branch.

Overdue tasks will automatically be moved to the 'Overdue' list if it has not yet been marked as complete and it has passed the due date. All assigned members will also receive an email notifying them of the overdue tasks.

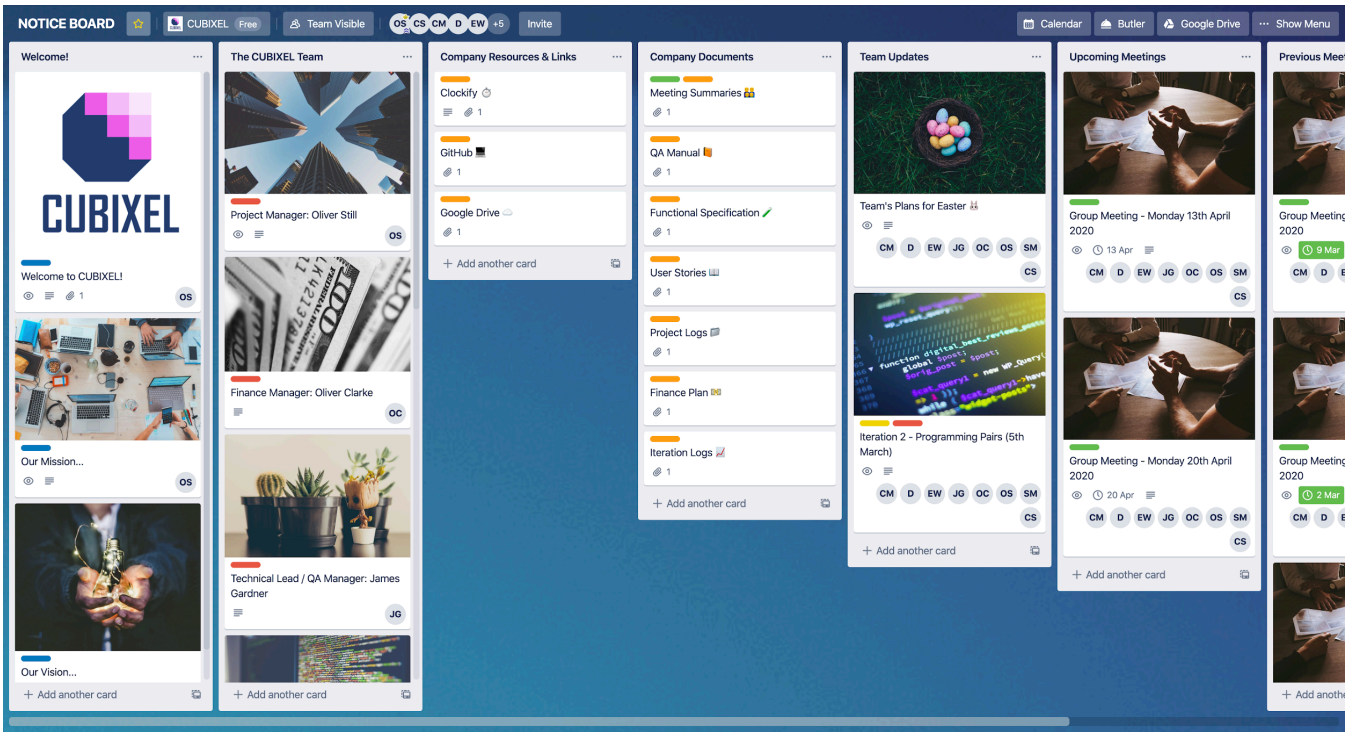


Figure 3 - Image of the 'Notice Board' workspace in Trello used by CUBIXEL for centralising and managing administrative tasks.

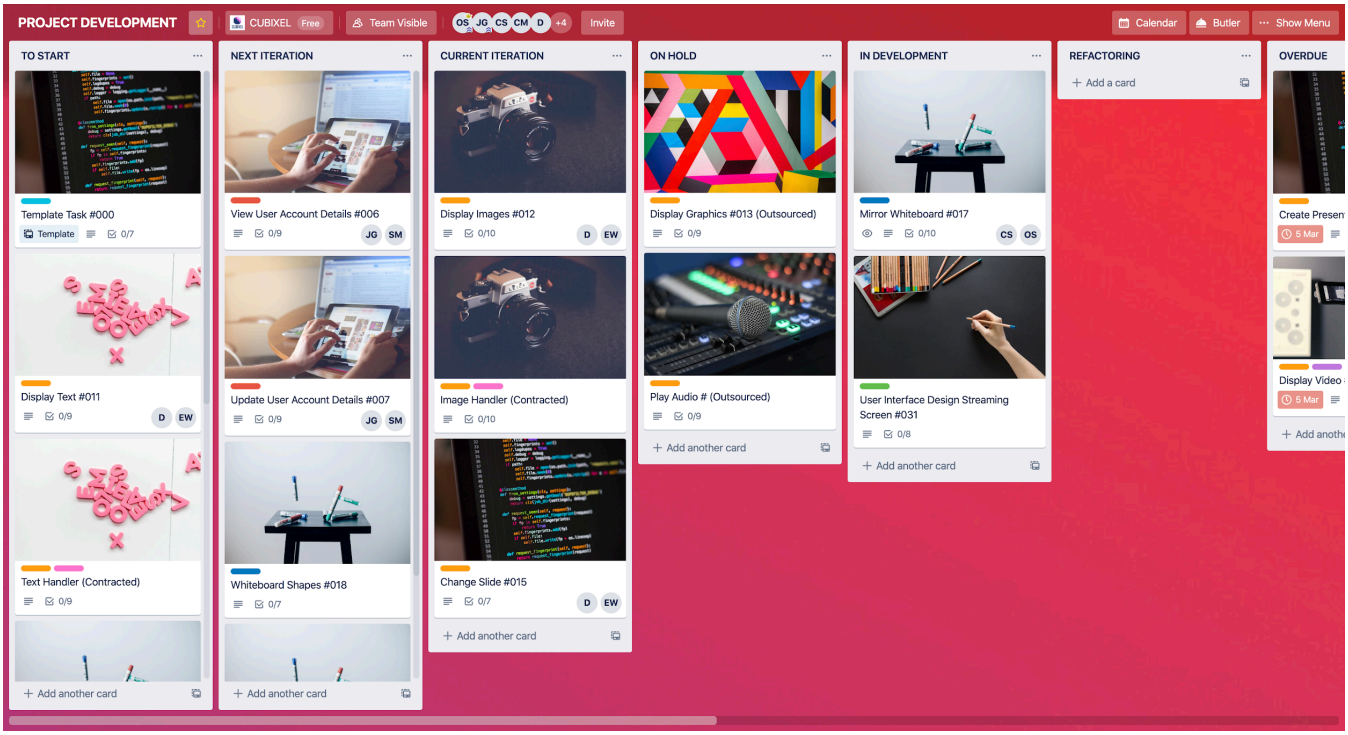


Figure 4 - Image of the project management workspace Trello used by CUBIXEL for organising and tracking User Stories.

4.1.3 Time Tracking

All team members are expected to be at the office from 0900 to 1700 (unless otherwise discussed with the project manager). This time includes entitlement to an hour unpaid break for lunch.

Time spent on projects are tracked via the third-party application Clockify. Using your university email address issued, you will be able to sign in to Clockify to view your respective project workspaces.

Within the application, you will be able to clock-in and clock-out your time at the office or log your hours working outside of the conventional office hours. The time tracked is additionally categorized depending on the type of work on the project.

These hours are then assessed by the finance manager to divide out the hourly pay of each team member.

4.1.4 Risk Management

Risk is assessed before and monitored during development by the project manager. However, risk is not solely the project managers responsibility and should be addressed by all team members. Whereby if a team member foresees a potential risk, they should make the project manager aware and discuss their concern. To reduce risk and help a project flow an iteration schedule is used.

Risk analysis involves the identification of a risk, the assessment of its importance and the evaluation of whether the risk level is higher than the risk that could be accepted for the project. In case a risk exceeds the acceptable levels, a risk analysis activity is instantiated by the project manager, potentially including team members if necessary. The analysis defines the required actions in order to reduce the risk to acceptable levels.

Risk management involves the planning of the required activities to handle the risk. This involves the distribution of resources, the evaluation of the results and current progress made within development.

4.1.5 User Stories

Projects should use User Stories to help define work to be done. Stories are short, one to two-line descriptions of work to be done written in a customer-centric style, and are a placeholder for a physical, face to face, conversion with the on-site customer. User stories should be stored within a 'Project User Stories' document.

They should contain:

- A couple of lines describing the work to be done, which should follow a "Who?" What? Why?" format.
    - Who the work is for (e.g. customer, user, another system)?
    - What will the story do (the function of the work or interaction of the systems)?
    - Why is this valuable to the project/customer?
- A list of customer-centric requirements.
    - A checklist that determines all the parameters of a User Story and are used to determine when a User Story is completed and working.
    - Used as a catalyst for test cases and should be testable.
    - Provides a detailed scope of the requirement, which helps the team to understand the value and help the team to slice the user story horizontally.
- A list of customer-centric tests.
    - What needs to be done to ensure and validate that this story is working correctly.
- Development time estimates.
    - Determines the level of resources and productivity compared to other user stories.
    - Used in estimating the overall project labour budget.
    - Measured as "ideal days" (eight hours of work per day per person).
- Value estimates / Priority.
    - Determines the value of the story to the customer.
    - Categorised between high, mid, low, and optional priority.

It is advised that value estimates are done prior to time estimates as the time requirements of a story can influence groups opinions over the value of a story. A story that requires a large amount of effort does not necessarily mean that story is valuable.

Remember to ask about the Non-Functional Requirements (NFR) of a project when speaking to the customer, such as time requirements for a process. These NFRs should be included in a user story. story sizes should be chosen so that the team can complete between four and ten of them each iteration.

Prioritisation

User stories are simply categorised by high priority, mid priority, low priority, and optional. These categories are relative to the importance of the program's base functionality, program flow, and the value to the customer. Tasks may change priority depending on the current state of the project and the project deadlines. The client may also request tasks to change priority based on their level of satisfaction of certain features.

Time Estimates / Ideal Days

All user stories need to be assigned a rough estimate of a time frame before development can start in order to map out the project timescale. Ideal days are defined as a full working day as a company (9am to 5pm), or 8 hours of programming. This gives an indication of the time frame and intensity to fully implement the respective user story into the project, as well as enabling us to budget for future iterations for a project's completion.

The number of ideal days must be agreed or discussed in advance in a weekly team meeting with each team member's guess on a number taken into consideration. This method allows multiple experts to give cross-disciplinary expertise and encourages discussion and justification to give a realistic timeframe. Additionally, for more vague specifications, an average can be taken to give the best estimate possible.

Trello Story Cards

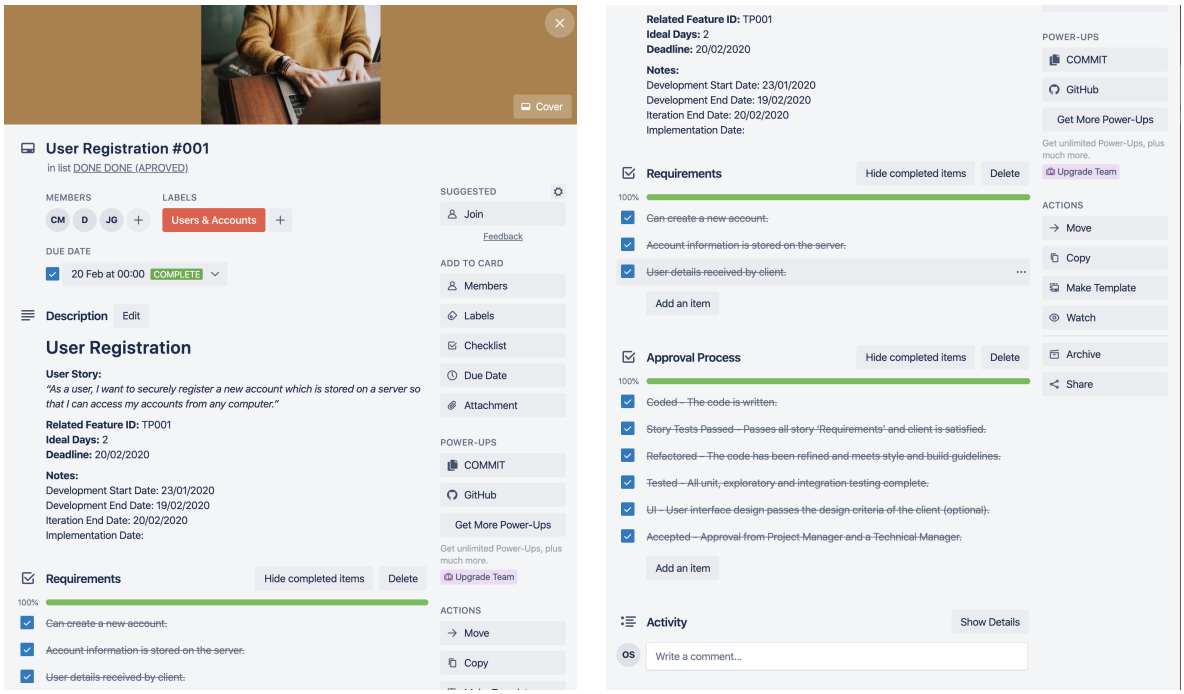


Figure 5 - Example of a completed user story card on Trello. It details the tests that need passing and all stages of the approval process must be completed before the card can move to 'Done, Done'.

4.1.6 Iteration Level Flow

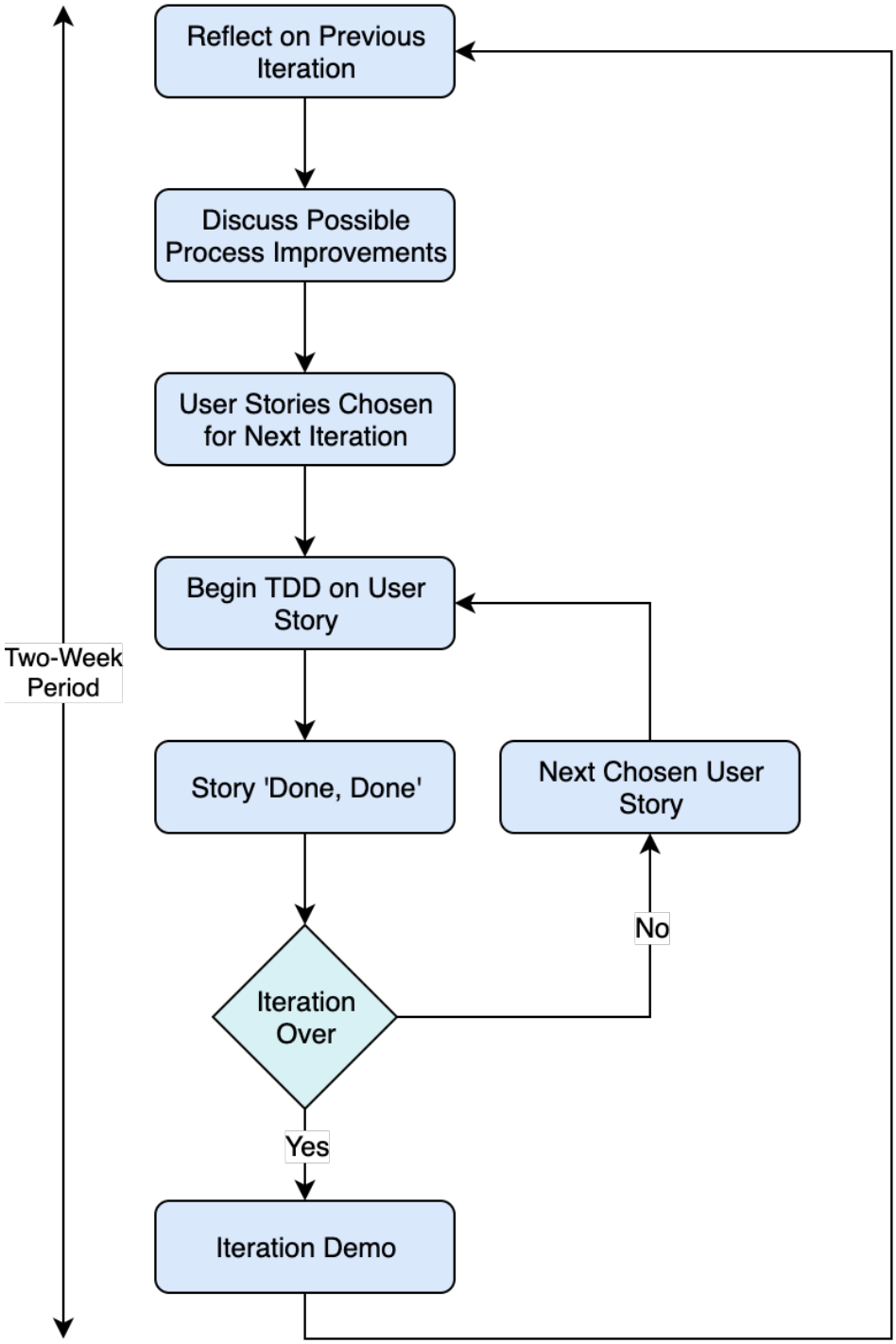


Figure 6 - Iteration level flow chart detailing how each two-week iteration should be broken down.

4.1.7 Git Commits and the Iteration Cycle

The Git repository management graph is shown below. This shows how the Master branch is always a fully functioning and customer presentable version of the program. The Development will always be up-to-date from the master for pulling new branches to develop new features. The Testing branch will be created at the end of an iteration, and will be used for testing and integrating new features for the product, with regular fully functioning commits to the master branch. The design of multiple new features simultaneously can be done using branches off the development branch before being committed once the design is complete.

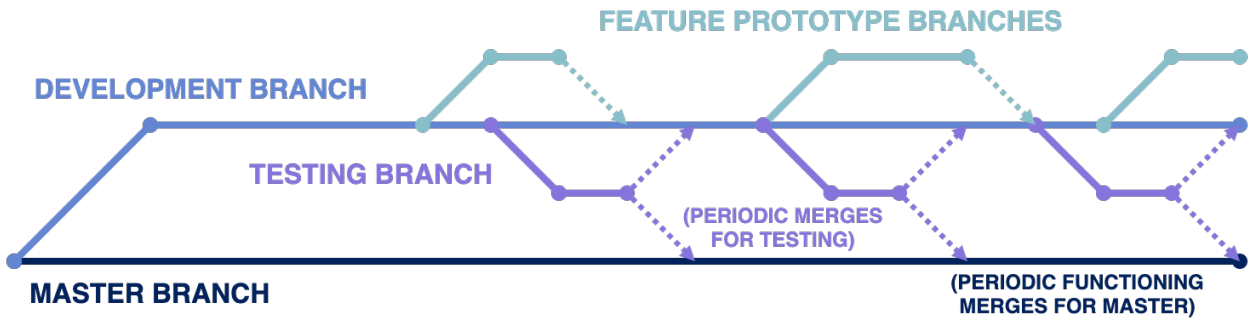


Figure 7 - General branch structure of a project's git repository in its development.

Each iteration will last two weeks with the exception of the first iteration of the project which can be extended up to four weeks in length at the Project Manager's discretion. This is to allow time for fundamental elements of the software to be worked on providing the second iteration with a strong foundation on which to deliver real customer value. It also provides an opportunity for the team to identify areas that will require further research and set aside time to complete that research.

4.2 Collaborating

4.2.1 Customer Involvement

We will have an on-site customer to help in decisions relating to prioritised stories, required features and to contribute knowledge of the field we are producing software for.

The on-site customer should be available to be asked questions at any time during development and is considered an integral part of the team.

4.2.2 Stand-Up Meetings

CUBIXEL team members will schedule and attend stand-up meetings, during which the whole team will assemble into a circle. These are usually called by the project manager, alternatively in their absence or urgency, by the lead technical developer. During the meeting each person will be allocated approximately one minute to briefly describe new information that the team should be aware of; each member should aim to answer three questions:

- What work have I recently completed?
- What work am I planning to complete?
- What problems are hindering me from completing my work?

To ensure the brevity of a stand-up meeting, the team could consider timeboxing the meeting by setting a timer appropriate for the number of attendees, team members could also consider preparing their statements before the stand-up meeting.

The use of stand-up meetings should not be considered an alternative to general communication between team members; members should communicate issues as soon as they arise rather than wait for a scheduled discussion.

4.2.3 Coding Standards

During the company's first project, further refinement of the coding standards that we will all adhere to should be defined. This should mostly be complete after the first iteration as the first user stories are reviewed and considered 'Done Done'.

Syntax

The general syntax of code produced by the CUBIXEL is based directly off the Google Java Style Guide. Any variations on this will be updated on the company syntax document and any suggestions on alternatives should be directed towards the Technical Manager for consideration.

Operation System (OS)

CUBIXEL will use both Windows 10 and macOS as default operating systems.

Integrated Development Environment (IDE)

The IDE that all software development should be done within is IntelliJ Ultimate, licenced by JetBrains.

Tools

All projects will use Maven for handling dependencies.

All projects will use LOG4J along with SLF4J for logging.

All projects will use Java SE 11.0.5 (LTS) as the default JDK.

All projects will use JUnit 5 and Mockito for testing.

File and Directory Layout

Standard Multi-module Maven project structure.

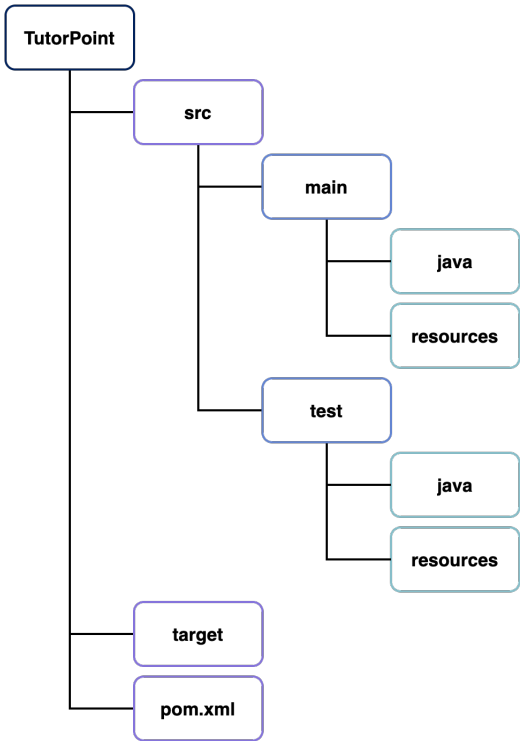


Figure 8 - Example of a single modules file structure. Note all tests are in a testing subfolder, with a layout mirroring that of the main folder.

Logging

Logging using Log4j should be output to both the Console and a File for ease of testing and fault finding once software deployed.

Standard Log4j XML shown below:

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
           <pattern>%date [%thread] [%-5level] %logger{40} - %message%n</pattern>
        </encoder>
    </appender>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>logFile.log</file>
    <param name="Append" value="false" />
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
      <pattern>%date [%thread] [%-5level] %logger{40} - %message%n</pattern>
    </encoder>
  </appender>

    <logger name="cubixel.tutorpoint" level="DEBUG"/>

                                        Logger Level

    <root level="INFO">
        <appender-ref ref="STDOUT"/>
        <appender-ref ref="FILE"/>
    </root>
</configuration>
```

All Loggers should be instantiated using the name of the Class the logger is being used in as it's input string:

```
private static final Logger log = LoggerFactory.getLogger("ClassName");
```

Level	Description
DEBUG	Relatively detailed tracing used by application developers. The exact meaning of the three debug levels varies among subsystems.
INFO	Informational messages that might make sense to end users and system administrators, and highlight the progress of the application.
WARN	Potentially harmful situations of interest to end users or system managers that indicate potential problems.
ERROR	Error events of considerable importance that will prevent normal program execution, but might still allow the application to continue running.

You might want to increase the logging level of a logger to diagnose or debug a problem.

4.3 Developing

4.3.1 Test-Driven Development

We have opted to use test-driven development (TDD) while writing our program in order to ensure a higher quality of code for our final product. TDD is a method of writing code that aims to reduce the number of bugs present in production code by fundamentally changing the process by which the code is written. Instead of writing an entire program then attempting to debug it, TDD requires the programmers to first think of a test that the feature being added will need to pass. The minimum amount of code required to pass this test is then written, and once the code is able to pass the test refactoring can be done to ensure the code is easy to build on.

Adding more and more tests until complete functionality can be measured by simply passing all the tests means that the simplest solution is normally the one implemented, reducing the risk of bugs. As the tests are only added one at a time and the code must pass all existing tests before another test can be added, there is no risk of a new feature breaking a previously implemented one.

By programming in this way, we can break a complicated program down into manageable chunks, called “User Stories”, and implement one by one so that there is always a functioning program which is constantly improved. A comprehensive list of tests to pass means that any errors can be detected as soon as they are introduced, making them significantly easier to correct than if they had gone unnoticed.

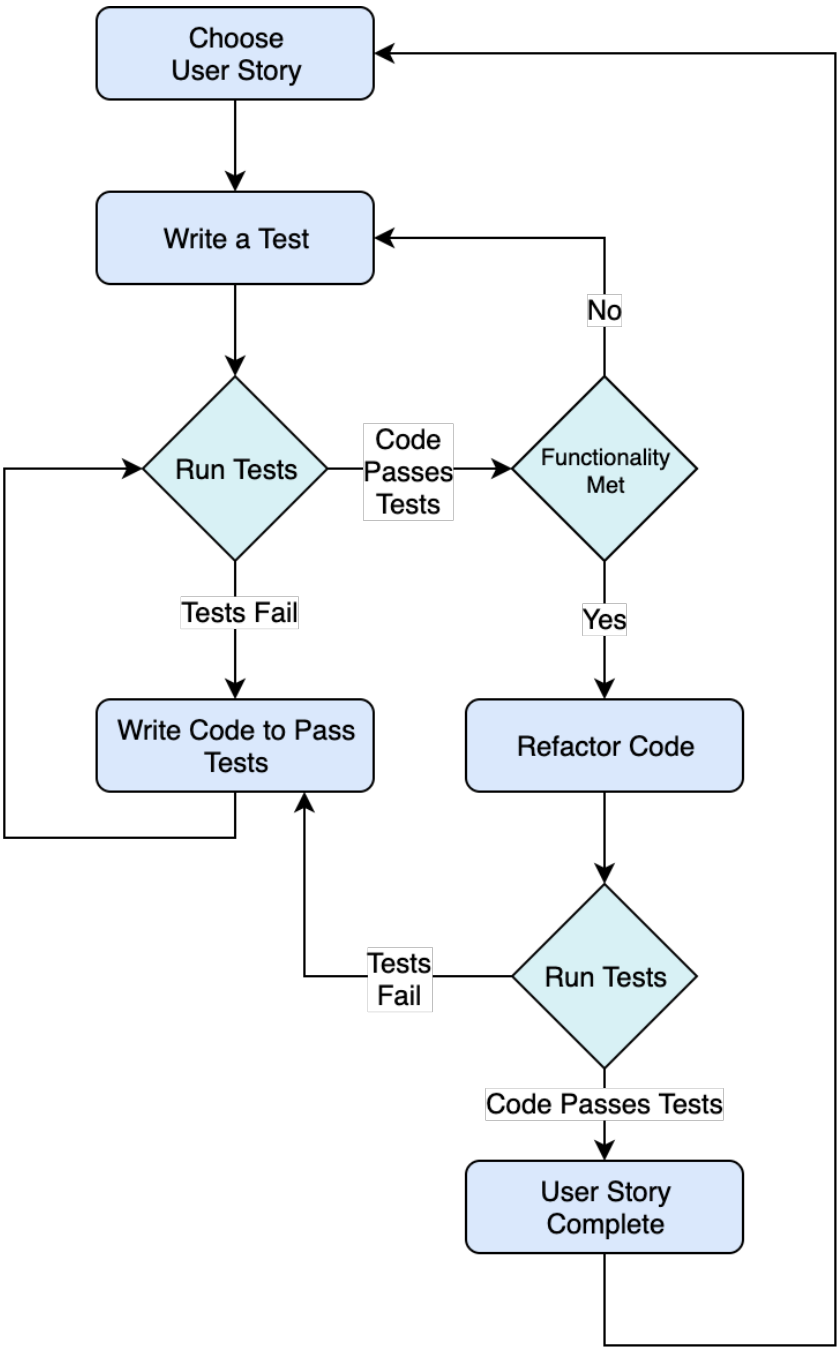


Figure 9 - Flow Chart of the Company's Test-Driven Development Process.

4.3.2 Refactoring

Development on the codebase should always be an iterative process. In the case of refactoring this will not change the function of the code, it will change how the code achieves its function whether it be rewriting it to be clearer or more efficient.

As refactoring requires an existing knowledge of the codebase, it works best when a less experienced programmer is paired with a more experienced programmer who has a good knowledge of the code being refactored.

Refactoring should only be done on code that is already passing tests, if the code isn't passing then the focus should be on functionality not on implementation. The first step should be improving code reuse, look for blocks of code which should be converted to functions. Then check for adherence to the company's coding standards.

The final step in refactoring is clean-up, looking for methods or variables which are no longer used due to the changes, removing old comments which are no longer relevant and other artefacts left behind due to your changes.

4.3.3 Exploratory Testing

At the end of each iteration, or after all user stories are ready to be integrated with the main branch, there will be a session of 'Exploratory Testing'. This will involve creating a charter, a statement to guide the aim of the tests and describe the aspect of the program that is to be explored.

Each completed user story to be integrated must first be pulled to the development branch. Next, the tester performing the exploratory testing will design a test on the overall program, execute it, and use the results of that test to decide what test to perform next. This cycle will then be repeated until the tester is satisfied that they have fully explored the relevant section of the software and fulfilled the charter.

During the 'Design-Test-Evaluate' cycle described above, the tester should keep notes describing what was tested, what the results were and any other observations that may imply incorrect function.

The purpose of the 'Exploratory Testing' is to find bugs and ensure the overall functionality is not broken, and as such, the tester should design tests to find weak points in the program. They should use test heuristics, varieties of tests that are likely to find the bugs in the code without exhaustively testing it. These heuristics could include:

- **Boundary Testing** - Inputting values that are just below, right on or just above a limit on an input field
- **Quantity Testing** - Testing how the program responds to having none, one or many of something
- **Input Sanitisation** - Ensuring that input fields cannot be used to input external code, e.g. SQL Injection.

While 'Exploratory Testing' is intended to find bugs, it is important that it is not used as a checkpoint to ensure that the code is as bug-free as possible. Instead, if bugs are found, this information should be used as feedback to the programmers to ensure that they are producing code that is as bug-free as possible, in line with the 'No Bugs' ideal (4.4.2). Testing must be completed within five days after the end of an iteration.

4.4 Releasing

4.4.1 Done Done

Once the code for a user story is written, the story can be considered 'Done'. However, this does not mean that the story is complete and finished. A story can only be finished, or 'Done Done', once it has passed these criteria:

- **Coded** - The code is written.
- **Story Tests Passed** - Passes all story 'Acceptance Criteria' and client is satisfied.
- **Refactored** - The code has been refined and meets style and build guidelines.
- **Tested** - All unit, exploratory and integration testing complete.
- **UI** - User interface design passes the design criteria of the client (optional).
- **Accepted** - Approval from Project Manager and Technical Managers.

After all criteria are met and reviewed, the code may be queued to be integrated into the master branch of the project at the next routine push request.

4.4.2 No Bugs

CUBIXEL strives to produce code that is theoretically completely bug free by attempting to write code that contains no bugs in the first place, rather than by using 'Regression Testing' to identify bugs that need to be fixed. This will be accomplished by:

- Using Pair Programming and TDD to ensure that good programming practices are used.
- Working closely with the on-site customer to ensure that code is not just bug-free, but also does what is desired by the customer.
- Take time to clean up sections of code that are likely to cause bugs to appear, such as shortcuts or designs that have not held up over time.
- Keep designs simple and refactor often during development to keep software from becoming a 'bug breeding ground'.
- Fix bugs quickly to make them easier to fix.
- Analyse the root cause of the bug, then learn from it to prevent similar bugs in the future.
- Use Exploratory Testing on any 'finished' sections of code to find any bugs that may have been missed, then, as before, learn from these bugs to prevent them in the future.

4.4.3 Continuous Integration

CUBIXEL uses a continuous integration approach to avoid large portions of code being developed and leading to a long and difficult testing and integration process to get multiple systems to work together.

All user stories in development/testing are pulled from the development branch into their own dedicated branch for implementation.

Once the code is complete, the branches can be merged where the feature is then ready to be integrated into the master branch at the next iteration date.

All user stories should be integrated into the main branch at the end of each iteration. Ideally at the end of each iteration an executable/installable fully functioning version of the master branch should be available to be shown to the customer in the 'Iteration Demo'.

4.4.4 Collective Code Ownership

The development of the project should not be critically impacted by the loss of a single person, ownership of the codebase should be spread collectively amongst the developers working on it.

The core concept of collective code ownership is that it doesn't matter who originally wrote the code you're working on, it is now your code, if you see a bug or an improvement to be made, then do it. This allows the code base to improve, as your working solution may not be the best solution; collective ownership allows another team member to build off of your solution. Collective ownership is not an

excuse for no ownership, all team members have responsibility for the codebase. This risk can be mitigated by having a focus on constantly refactoring the code as well as having a robust continuous integration pipeline.

Pair programming will help all team members build their knowledge about areas of the codebase they may be unfamiliar with. If you see someone taking on a task that they are inexperienced, offer to pair with them to share your knowledge about that part of the codebase.

4.5 Deliverables

4.5.1 Iteration Demo

An iteration demo should be shown to the team and customer at the beginning of each iteration. This should take no longer than ten minutes and be used to demonstrate the current state of the software.

This informs the customer of the current progress and allows an open discussion between all involved over features and issues.

The results of this discussion should be used to inform the choice of User Stories for the next iteration.

4.6 Ideals

4.6.1 Pair Programming

The majority or production code should be produced in pairs. One person in the pair will be the ‘Driver’. They will be programming and have direct control over coding the solution to a task or problem.

The second person will be the ‘Navigator’. It is the Navigators responsibility to help the Driver focus on the task. This can involve thinking about what task to work on next or how this code fits into the bigger picture. Navigators should not rush to point out small syntax errors in the Drivers code, this would get annoying quickly. Navigators are there to help the Driver work efficiently.

It is advised that the pairing partners are switched occasionally to allow people to switch focus onto other tasks, get new perspectives on problems and to integrate well with the entire team.

Pair programming should also be used to ensure all members follow the coding standards as detailed in section.

Small portions of code, code adjustments such as minor syntax changes or Spikes into new technologies are okay to be done without a pairing partner.

4.6.2 Root Cause Analysis

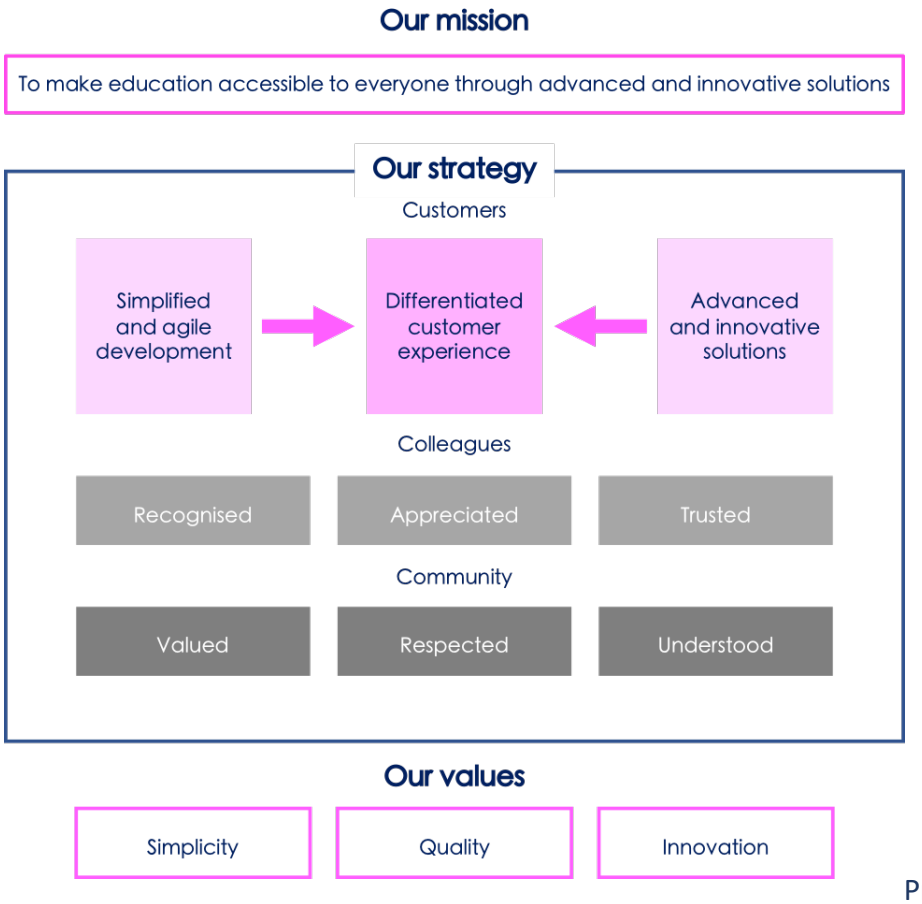
CUBIXEL will utilise root cause analysis, an approach for identifying the underlying causes of an incident so that the most effective solutions can be identified and implemented. In order to identify the root cause of a problem, the person investigating should ask “why” at least five times in regard to what is causing the problem. Using this method should reveal a specific and easily targetable issue that can be addressed systematically to achieve a solution.

Root causes will be discussed and addressed during team meetings. Relevant team members should coordinate to find appropriate solutions to the root causes; project teams with control over the root cause should gather to discuss thoughts and cooperate in fixing the issues raised. Team members should avoid over-applying root cause analysis; fixing a root cause of a process may add overhead. Before making any changes to a process, team members should consider whether the problem is significant enough to warrant the overhead.

4.7 Marketing

4.7.1 Vision and Mission Statement

The vision and mission statement for the CUBIXEL company will be utilised by the CUBIXEL marketing team in order to direct marketing initiatives and actions during the product development process, and after product completion and release. The vision and mission statement will additionally be used as a cross-check, enabling the CUBIXEL marketing team to conclude whether marketing initiatives and actions are in line with the vision and mission statement, or if they need to be reconsidered and altered.



4.7.2 Product Analysis

The CUBIXEL marketing team will undertake extensive analysis of the product being developed by the company, such analysis will consist of an overview and examination of the relative strengths, weaknesses, opportunities and threats. Strengths highlight the successful aspects of the CUBIXEL company and the product it is developing, which assist in distinguishing from competitors; they are integral, bringing clear advantages in the engineering field. Weaknesses are areas in which the CUBIXEL company and the product it is developing lack in competitive strength in comparison to competitors, potentially resulting in the hindrance of strategy and product mission success, the analysis of such weaknesses will enable CUBIXEL to utilise its strengths and future solutions to combat them. Opportunities are openings for the CUBIXEL company to improve and grow the products it develops, the ability to spot and exploit opportunities can make a huge difference to a product's ability to compete and take the lead in its market; CUBIXEL will utilise its strengths to capitalise on highlighted opportunities. Threats are factors that can externally and internally affect the success of the mission established for a product in development; threats cannot be controlled, however, they can be identified and their effects mitigated.

4.7.3 Market Analysis

The CUBIXEL marketing team will carry out substantial research into the market context in which the product being developed resides. By understanding market trends in the product's industry and in others that are associated, CUBIXEL can take advantage of opportunities as they arise and act quickly to reduce any risks to development or deployment of the product. To achieve such research, primary and secondary market research methods will be utilised. Utilising secondary research methods will involve seeking out existing research and data sources that can be applied to the project. A majority of secondary sources will be sourced from the internet; examples include demographics and market research compiled by similar companies/ industries and reports issued by research institutions. In the possibility that secondary research methods produce insufficient data, the CUBIXEL marketing team will utilise primary research methods. Primary research methods will be carried out personally by the CUBIXEL marketing team; the predominant method that will be used is survey, however, field trials and focus groups may also be utilised if necessary.

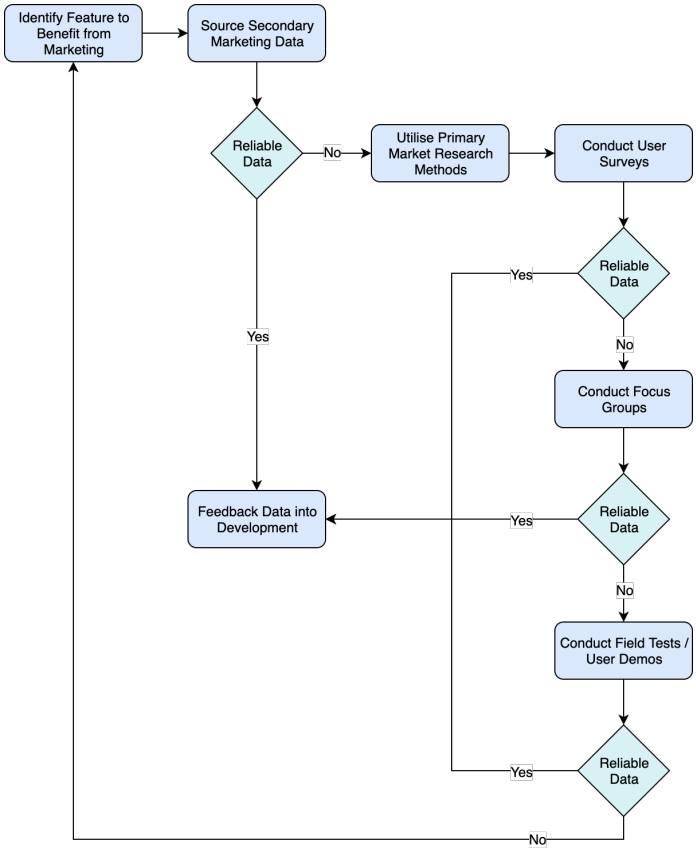


Figure 10 - Breakdown of the marketing process within development.

4.7.4 Marketing Strategy

The CUBIXEL marketing team will utilise the vision and mission statement of the CUBIXEL company in order to establish a marketing plan and strategy for the product it is developing. The product marketing strategy will detail imperative information, consisting of the product target market, the product target demographic/ persona, the marketing initiatives for the product, and the marketing channels that will be utilised by the marketing team to achieve marketing initiatives; further detailing how such channels will be used. Defining a target market or industry is crucial to the initial success and growth of the brand and product developed by CUBIXEL; such definition impacts advertising, as well as customer experience, branding, and business operations. In order to assist the successful implementation and operation of a marketing strategy, it is important for the CUBIXEL marketing team to clearly define marketing initiatives

for its developed product. Stating marketing initiatives and how we intend to fulfil them enables us to devise marketing strategies more effectively, while allowing us to monitor progression and measure success. The ability to measure the success of our strategies grants us the additional benefit of being able to make alterations where necessary to improve strategy effectiveness. To successfully execute our marketing strategy for the CUBIXEL developed product and hence achieve marketing initiatives, it is important to explore the marketing channels available to the CUBIXEL marketing team that will assist in producing the best marketing outcomes.

4.8 User Experience

4.8.1 Graphics

The CUBIXEL user experience team will visually design UI elements (icons, font-typing, etc) to the following criteria:

- **Clarity** - Will it be clear to the user? This concerns if an element can actually be seen by the user properly, if it's unobscured of other elements, if its colour composition makes it readable or not.
- **Aesthetics** - Does it look nice or at least satisfactory to a majority of users? What individual users enjoy visually can vary wildly, so making the style accessible to the largest number of users is critical.
- **Appropriateness** - Is it the correct level of visual complexity? A UI element should only be as developed as it needs to be (stylistic considerations are included in this upper bound) but should also not be so simple that it might be obscure to the user.
- **Consistency** - Does it follow/fit in with the visual style of the elements that are already in the UI, and should it? This means looking at the shape, colour, and the size of the element being added; then comparing it to content that has already fulfilled the design criteria. The key is to have contrasts within the UI where the user's attention needs to be drawn (i.e. having an element be clearly different in colour than its surroundings, if that element doesn't relate to surrounding elements or has sufficiently different functionality) but also uniformity for elements that serve a common purpose.

4.8.2 Interaction

The CUBIXEL user experience team will design how the user interacts with UI and its components to the following criteria:

- **Practicality** - Does it serve to improve or optimize user experience? If the element doesn't help direct the user, make the UI more understandable for the user, or otherwise improve utility to a user; then that aspect is very likely frivolous, surface level, and unnecessary.
- **Clarity** - Is its function clear enough to the user? An element on a more surface level of the software's UI (the base features needed to use the application for its intended purpose) should be very intuitive in terms of its function. An element in a lower level of the UI (more complex features that aren't necessary to be accessed by the average user) can afford to have more 'explanation' to its function.
- **Flow** - Does it encourage fluid navigation of the application? An application that has poor flow and is clunky will make users less likely to use the software.
- **Enjoyability** - Does it make utilization of the software satisfying? Quite a similar condition for good interaction design to flow, however it's more general and does not just concern itself with efficient transition to other sections of the UI. Enjoyability is achieved through good design of both visual and interactive aspects of UI elements.

4.8.3 Project Specific Considerations

In the scenario that CUBIXEL takes on a project that has other front-end, user experience aspects (i.e. a social media application that has sociological implications, or an application that works in conjunction with peripheral hardware) then the user experience team will perform research on these additional

considerations; they will then build upon the existing graphics and interaction methodologies with these factors in mind, as well as creating new design methods and models concerning the elements (with potential for that documentation to be used on future projects with shared properties).

4.8.4 Design Within Iterations

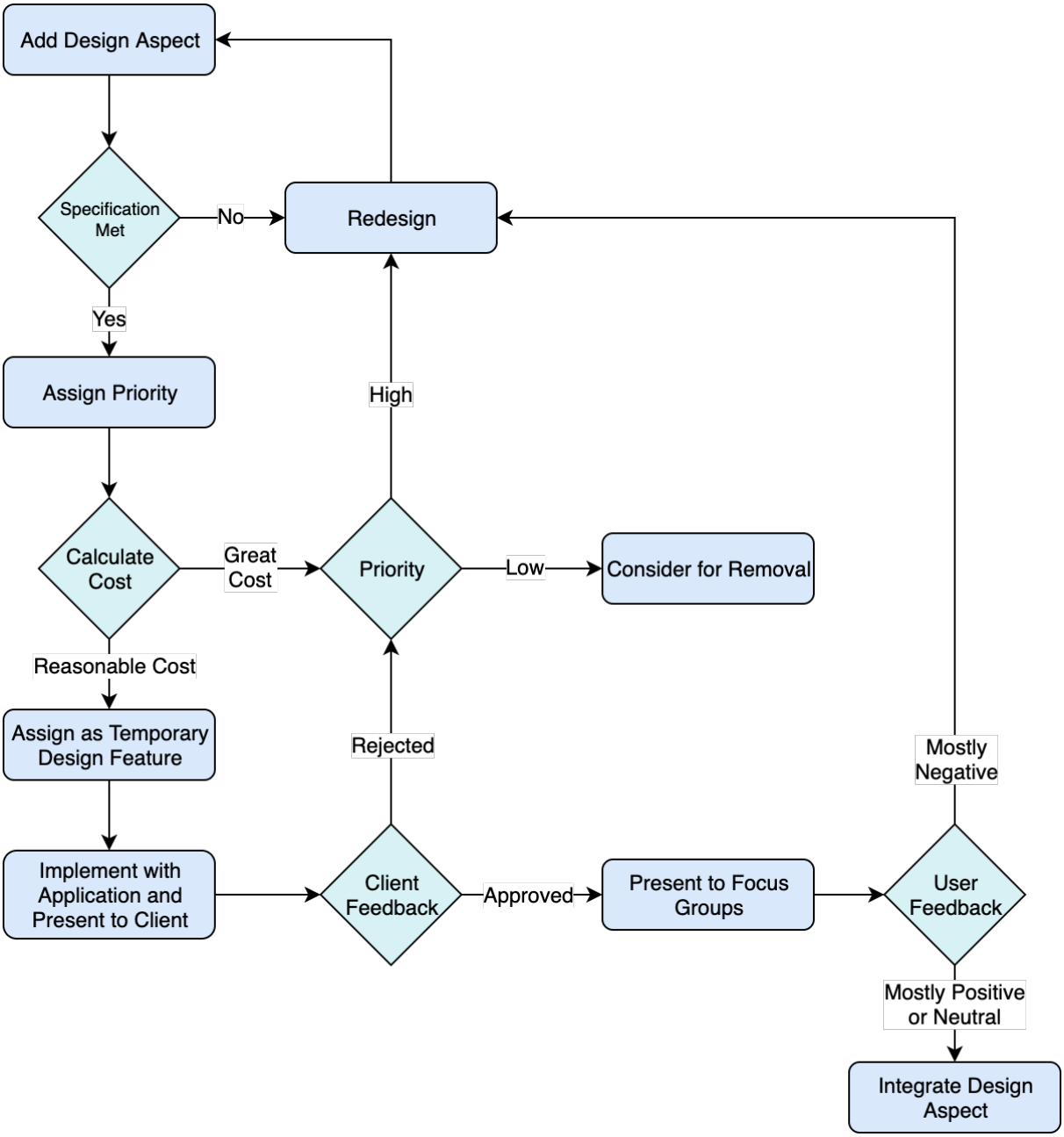


Figure 11 - Breakdown of the design process within development.

5. Facilities and Safety

5.1 Policy

The office staff are provided with a personal desk with enough space for two people to comfortably sit and work together and a comfortable secure environment in which to perform their work. All facilities meet the standards of the Workplace (Health, Safety and Welfare) Regulations 1992.

The office is designed to be an open and enjoyable environment supporting and encouraging open discussion and an agile development philosophy.

All personnel receive basic training on maintaining the standards of the office and of the office evacuation plan on the day of induction.

5.2 Facilities

The office has several areas designated for specific purposes; for example, open office space, storage, kitchen, WC's, chill-out area, meeting rooms/conference rooms.

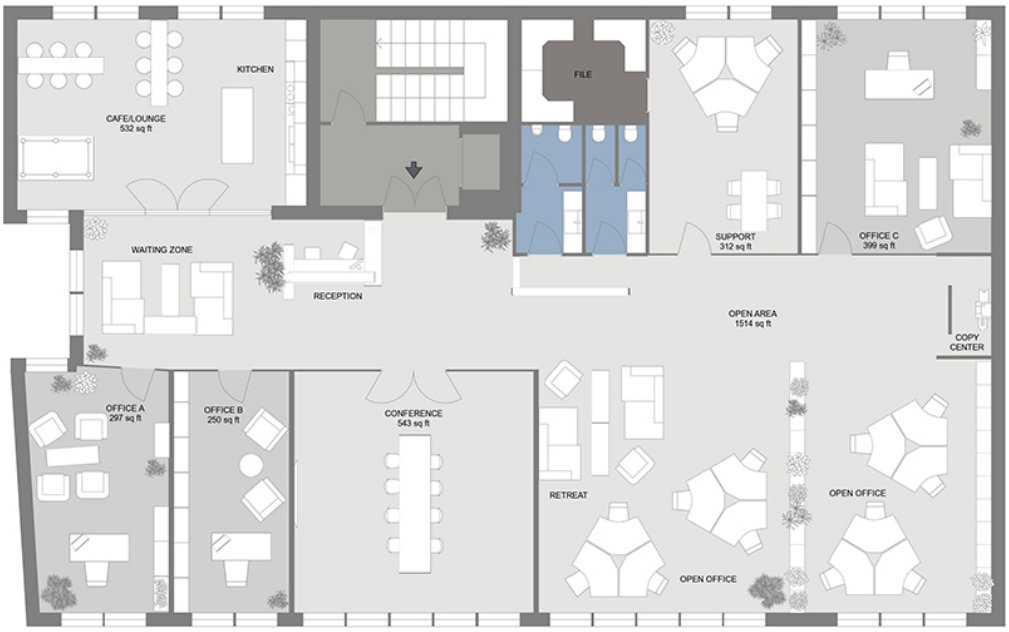


Figure 12 - The CUBIXEL office floor plan.

5.3 Security

It is the responsibility of all members of the CUBIXEL to keep the office a secure environment. Non-permanent staff will be given ID Cards to identify them within the building. All windows and doors should be locked by the last person leaving the office and all lights should be switched off.

5.4 Working Environment

The working environment should be clean and uncluttered. It is the responsibility of all employees to keep their desk areas clean and tidy and to leave any area they are working in, in a better state when they leave than when they found it.

5.5 Waste Disposal

The company aims to reduce waste as much as possible. Employees are encouraged to reduce, reuse, and recycle personal and company waste. Every area of the office contains recycling points and food items should be cleaned prior to being put into recycling. Food waste should be disposed of correctly in the marked food waste bins. Bins are emptied each morning by the cleaning staff.

6. Equipment

6.1 Policy

All staff are provided with the necessary equipment to perform their role. Equipment is chosen to be of a high standard and provide a comfortable and productive workspace. Each desk will be provided with a single computer, two monitors, two chairs, a keyboard and a mouse to encourage pair programming. All staff must use the computers provided by the company.

A small number of laptops are available for working from home. However, this is discouraged for software engineers as it would remove the ability of pair programming.

6.2 Selection of Equipment

Selection of the equipment is decided between the CEO, Financial Manager and Technical Managers. Any suggestions for alternative equipment should be quickly highlighted to any of the above three.

6.3 Equipment Inventory

An inventory is maintained of all equipment currently owned by the company. This is updated by the Office Administrator as and when needed. The document is checked in audits done throughout the year.

6.4 Maintenance and Repair

The responsibility of maintenance and repair of any equipment owned by CUBIXEL falls on the company. Employees are responsible for treating company equipment in a reasonable fashion.

6.5 Decommissioning

Decommissioned equipment will be disposed of by CUBIXEL in an eco-friendly manner via appropriate disposal sites.

7. Purchasing

7.1 Policy

The company's policy on purchasing items and services are regulated by the finance manager and overseen by the project's respective manager. Requests must be formally written and sent to the finance manager for approval. If approved, the payment may proceed and a receipt for the company will be produced. General consumables must not be requested to the finance manager, but instead must be noted on the 'Last One Board'. The company also incorporates purchase authorisation limits depending on the role within the company. The limits are subject to change at any point depending on a project's budget and timeline. Details of the purchase authorisation limits may be acquired from the finance manager.

7.2 Consumables

The company maintains a 'Last One Board' that is checked at the beginning of each week by the Office Administrator. Should any consumable items (e.g. pens, board markers, notepads, etc.) be down to the last item, this should be marked on the 'Last One Board' document by the member of staff who noticed. New consumable items will then be ordered.

7.3 Selection of Providers

Providers are selected based on their ethical ability to provide and support their service. If a provider is found to provide unethical practices to their employees or services, alternative providers that match with our standards will be utilised instead.

Providers will be selected based upon cost. Any technical requirements should go through the consultancy of relevant members of the management and be discussed as a team to decide the best product.

7.4 Stock Management and Inventory

Company stock is recorded and maintained by the executive receptionist. Low stock of any office or company resources must be noted in the 'Last One' document located at the front desk. A stock order will then be requested by the executive receptionist to be confirmed by the finance manager.

7.5 Supporting Documents

Last One Board Template can be found in section 11.

8. Personnel

8.1 Policy

CUBIXEL is committed to treating employees fairly and with respect. We conform to the Equality Act (2010). We maintain a safe, non-hostile work environment. We have a policy of zero tolerance for harassment and bullying in the workplace. Any non-conforming behaviour will result in write-ups, and eventually firing.

8.2 Recruitment

When recruiting, it is CUBIXEL's policy to ask all candidates, both internal and external the same base interview question set, to avoid discrimination issues.

8.3 Personnel File / Health File

Information it will contain:

- Name and Residential Address
- Contact Mobile Number
- Bank Details
- Medical Notes / Emergency Details

8.4 Integration

All new employees will be assigned a Mentor (a senior member of the team they will be involved with) and a Buddy (a direct introduction to one member of their team) to provide an avenue for questions and advice throughout their induction period. All new employees must complete a 'New Employee Induction Form' at the end of the first week, first month and first three-month periods of their time at the company.

8.5 Training

New employees will be trained in CUBIXEL's development methodology and coding standards by their assigned Buddy.

8.6 Staff Competency

Staff are required to be competent enough to deal with their responsibilities. CUBIXEL will carry out yearly reviews to assure competency. If a staff member is found to be under-performing, they will first be warned of their conduct. If the employee is still found to be lacking, they will first be sent for a training course in the relevant area, with any prolonged inadequate performance resulting in disciplinary action. This may include being subject to a personal improvement plan (PIP) which will specify conditions which must be met in order to result in the employees continued employment.

8.7 Personnel Performance Appraisal

Personnel will be subject to a yearly Performance Appraisal. This may result in disciplinary action, as above, or bonuses. Non-management staff bonuses will be tied solely to the employee's performance. Management bonuses will be tied to both their team's overall performance and the performance of the company as a whole.

8.8 Non-Permanent Personnel

Personnel on a non-permanent contact will be subject to yearly performance reviews. Renewal of their contact will be dependent on their performance and the necessity of their role within the business.

8.9 Supporting Documents

New Employee Induction Form can be found in section 11.

9. Continual Improvement

9.1 Policy

CUBIXEL is always looking to improve. All employees are encouraged to share ideas and suggestions on any processes or ideals that the company can improve. Measurement of the efficiency and productivity of all processes should be done regularly, especially for newly integrated processes.

9.2 Quality Indicators

9.2.1 Company Agility

Using the Self-Assessment Quiz as detailed.

9.2.2 Client Satisfaction

A client representative will be on site at all times to ask questions and oversee development. The customer will be able to see the current functionality of the software each iteration at the 'Iteration Demo'.

At any of these demos the customer can request that a User Story (section of the program) is revisited or changed in some way. The number of revisions of code requested by customers at iteration demos can be used as a metric for the customer satisfaction with the design process.

If a customer does request a change/revision of a Story after it has been approved, root cause analysis should be used as to why this User Story passed the approval process. Perhaps further customer interaction is needed.

9.2.3 Value for Customers

By working closely with the customer during the development cycle we aim to provide a high value service. Any feedback received by any employee from a customer should be brought to management's attention to allow for improvements to be made where necessary or to highlight parts of the process which were more than adequate.

9.2.4 Employee Satisfaction

A companywide pulse survey will be run quarterly allowing employees to comment on all aspects of the business. The results will be brought to the attention of management, allowing them to see an overall reaction to any changes made to the business as well as any general areas where improvement is needed. Internal surveys may be distributed at the leisure of the team leader or project manager.

9.3 Eliminating Waste

9.3.1 Small Steps

CUBIXEL will strive to program efficiently by using TDD to develop software in small steps.

9.3.2 Failing Fast

CUBIXEL will reduce code waste by ensuring that if a feature is not yet implemented, it will fail fast. This will ensure that there is no wasted code supporting unimplemented features.

9.3.3 Minimum Viable Product

CUBIXEL will first deliver a minimum viable product to the customer. This will reduce waste by allowing the customer a chance to review the product before any unnecessary development is done.

9.4 Seeking Technical Excellence

CUBIXEL is committed to seeking technical excellence by embracing the ethos of XP. Additionally, employees of CUBIXEL must be committed to continuous self-improvement.

10. Disaster Recovery

Serval external risks have the possibility of impacting the company's work dynamic. These risks include, but are not limited to office closure, server failures, and communication failures. Individual risks are also considered and assessed personally and reported to the Project Manager while constructing a Disaster Recovery Plan (DRP).

10.1 Risks

10.1.1 Office Closure

The office may be required to close with moderate or, in serious enough cases, immediate notice, requiring all employees to leave the workplace as soon as possible to another base or at home.

Risk	Likelihood	Response
Fire in the workplace or in the office building.	Low	All employees will be informed and compensated for any personal damage or items lost. Senior members of the company will switch to a temporary office space, while smaller teams may be asked to work remotely from home. Data will be recovered where possible.
Power outage in the workplace or in the office building.	Moderate	All employees will be asked to either continue work on personal devices or to travel home and work remotely. National grid engineers and the building resource managers will be informed. Data will be recovered where possible.
National lockdown due to pandemic.	Low	The company will follow government advice and events will be continually examined as they unfold. All employees will be required to work from home via a company VPN until further notice.

10.1.2 Server Failures

The company relies on the functionality of several third-party applications, including Google Drive for server storage, GitHub for repository storage, and other communication applications.

Risk	Likelihood	Response
Power outage at server hosts.	Low	Company will either switch to backup servers supported by the server host or switch to a local backup made by the company.
Third-party application servers fail.	Moderate	Company will determine the severity of the failures (temporary downtime due to updates or widespread server outage) and respond accordingly. In the case where Google Drive fails and becomes inaccessible, and physical backup will be uploaded to another platform or sent to all employees physically.

10.1.3 Communication Failures

The company uses various online communication channels to ensure clarity among teams and, most importantly, the client.

Risk	Likelihood	Response
A third-party communication channel fails.	High	Company will determine the severity of the failure (temporary downtime due to updates or widespread server outage) and respond accordingly. In the case where one communication channel fails, the company will momentarily switch to an appropriate alternative.
All communication channels fail.	Low	In the case where all communication channels go offline, all employees will be required to work from home on local material until further notice. A first point of contact or rendez-vous point will be defined for senior members of the company to adhere to, where future steps will be discussed.

10.2 Disaster Recovery Plan

In the case where all employees are required to work remotely, either in several smaller locations or with each member working from home, a Disaster Recovery Plan (DRP) will be put in place and briefed to all via a preliminary meeting or conference call.

On an employee by employee basis, individual risks and limitations will be examined and considered within the company's DRP. For example, limitations such as the employee's ability to work remotely will be assessed to consider what additional equipment they would need to efficiently work. These include elements such as the stability or speed of their internet connection, any needed office equipment like monitors and cables, and any required software for current projects that the employee may be working on.

As a general precaution, all employees are advised to have the standard OS, IDE, and additional tools defined in '4.2.3 Coding Standards' installed on an accessible home machine in the case of such a scenario.

Individual health risks will be analysed for each employee. These risks include the effect on mental health when working from home. Furthermore, in the case of a pandemic, the likelihood of exposure and, if exposed, the severity of the disease for each employee will additionally be considered. More personal information relating to any health risks from working from home will be disclosed at the choice of the employee, however will be considered confidentially in the creation of the DRP.

10.3 Working Remotely

If the unlikely scenario of all employees being required to work from home occurs, the following measures will be put in place as monitors and substitutes for the office environment:

1. The number of online meetings will be increased to two per week (Monday morning and Thursday afternoon) in order to ensure clarity and to detail any tasks to round off before the end of the working week.
    a. An 'Agenda' document containing all topics to discuss in the online meeting will be distributed to all attendees prior in order to allow for additions and alterations before the meeting is led.
2. Engagement between employees and communication channels are favoured to regular working hours (0900 to 1700).
    a. Employees are not expected to view or respond to messages or queries outside of these hours, but are free to do so in their own leisure.
3. All employees will be required to provide a detailed, daily log of completed work or current challenges at the end of the day in conjunction with their logged hours.
    a. Daily logs of each employee will be cross-checked to ensure efficient progress is still being made.
    b. Anomalies will be queried and assessed the same way as described in '8.6 Staff Competency'.
4. Current and future user stories of ongoing projects will be examined to gage whether the 'pair programming' ideal can still take place. Alternatives methods will be suggested, such as breaking down a user story into separate tasks for each member of the pair, or assigning the story solely to one employee.
    a. Employees will be encouraged to manage their assigned tasks between themselves.

11. Information Management

11.1 Privacy Policy

CUBIXEL (“we,'' “our”, “us”) collects and stores information about our employees and customers, this information is used for employment and contractual service purposes respectively. We are committed to protecting and responsibly handling your information.

Personal and identifiable information is information that we hold on you where you could potentially be identified. Most of the information we hold will be provided directly by you or be generated by us as part of our relationship with you.

11.2 Information Security

CUBIXEL is dedicated to ensuring the security of all information that it holds and is fully compliant with GDPR and any other relevant legislation in the countries we operate within.

1. Information should be classified according to an appropriate level of confidentiality. (PII vs Non PII information, Civilian vs Defence contracts, etc)
2. Staff with particular responsibilities for information must ensure the classification of that information. They must handle that information in accordance with its classification level and must abide by any contractual requirements for meeting those responsibilities.
3. Information should be both secure and available to those with a legitimate need for access.
4. Information will be protected against unauthorized access and processing
5. Breaches of this policy must be reported to the Information Commissioner’s Office by our data protection officer.
6. Information security procedure and the policies that guide it will be regularly reviewed, through the use of annual internal audits and penetration testing.
7. Information management systems used within the company will be appraised and adjusted through the principles of continuous improvement, as laid out in ISO27001 clause 10.

11.3 Confidentiality

CUBIXEL is dedicated to protecting the data of our employees and customers. Customer and employee data will be stored on separate encrypted servers, access to which will only be given to those who have a business need for it. Two factor authentications are mandatory when accessing personally identifiable information for either customers or employees.

Customer and employee data will not be shared outside of CUBIXEL with the exception of compliance with a legal request from the government of the country in which that data is stored, currently this is the United Kingdom. Therefore, data will only be shared due to a valid court order or a request made under the Regulation of Investigatory Powers Act 2000.

Accessing our internal network with an unapproved device (e.g. employee’s personal phone or personal laptop), inserting an unapproved storage device into a company computer or copying of any information from our internal network will be considered gross misconduct and will be grounds for termination and possible prosecution of the employee responsible.

12. Appendix A - Supporting Documents and Templates

12.1 Meeting Minutes Template

CUBIXEL

Minutes 2.0.2

Meeting # |Minutes

Meeting date | time *21/10/19 | 11:00* **| Meeting location** *Library LMO 225*

Meeting called by	Whole Team
Type of meeting	Weekly Scheduled
Facilitator	-
Note taker	-
Timekeeper	-

Attendees

Daniel Bishop (dmb537)
Oliver Clarke (odjc500)
James Gardner (jadg502)
Stijn Marynissen (sm2174)
Che McKirgan (cwjm501)
Cameron Smith (cs1869)
Oliver Still (os705)
Eric Walker (ew1150)

AGENDA TOPICS

Time allotted | *10* **| Agenda Topic** *Discuss Previous Meetings Actions* **|**

Discussion: To address any actions from previous meeting and check all tasks are progressing on track. Address any issues that have arisen over the week.

Conclusion: Closing

#	Action items	Person responsible	Deadline
	Topic 1	**Name**	**Date / Time**
	Topic 1	**Name**	**Date / Time**

Time allotted | *Time* **| Agenda topic** *Topic* **| Presenter** *Name*

Discussion: Conversation

Conclusion: Closing

#	Action items	Person responsible	Deadline
	Topic 1	**Name**	**Date / Time**
	Topic 1	**Name**	**Date / Time**

Time allotted | *Time* **| Agenda topic** *Topic* **| Presenter** *Name*

Discussion: Conversation

Conclusion: Closing

12.2 Assess Your Agility

AYA 1.0.0

Assess Your Agility

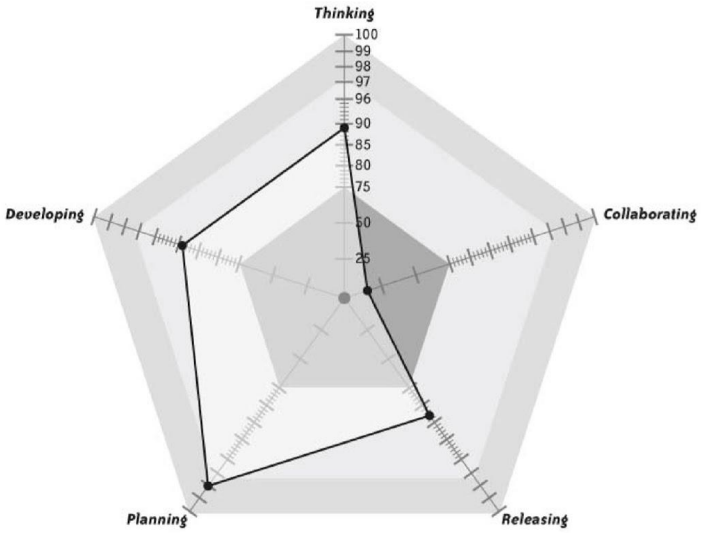
Document Description

This document is heavily based upon the suggested method for measuring team agility described in the Art of Agile Development. Each iteration it is advised that the team perform this test to provide a metric of agility and look for ways to improve.

Don't give partial credit for any question, and if you aren't sure of the answer, give yourself zero points. The goal should be to achieve the maximum score in each category. Any score less than the maximum indicates risk, and an opportunity for improvement.

- 75 points or less: immediate improvement required (red)
- 75 to 96 points: improvement necessary (yellow)
- 97, 98, or 99: improvement possible (green)
- 100: no further improvement needed

Example Completed Test



Note: The point values for each answer comes from an algorithm that ensures correct risk assessment of the total score. This leads to some odd variations in scores. Don't read too much into the disparities between the values of individual questions.

CUBIXEL

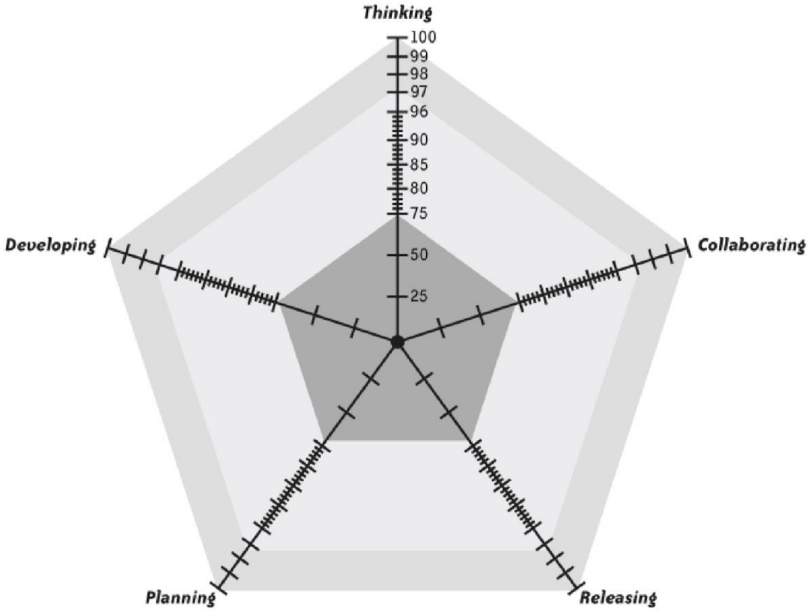
AYA 1.0.0

Blank Test Results Graph

Employee: _____

Date of Completion: _____

Supervisor Signature: _____



CUBIXEL

AYA 1.0.0

Self-Assessment Quiz

Planning

Question	Yes	No	Methodology Under Test
Do nearly all team members understand what they are building, why they're building it, and what stakeholders consider success?	25	0	Vision;
Do all important stakeholders agree on what the team is building, why, and what the stakeholders jointly consider success?	25	0	Vision;
Does the team have a plan for achieving success?	4	0	Release Planning;
Does the team regularly seek out new information and use it to improve its plan for success?	2	0	Release Planning;
Does the team's plan incorporate the expertise of businesspeople as well as programmers, and do nearly all involved agree the plan is achievable?	3	0	Planning;
Are nearly all the line items in the team's plan customer-centric, results-oriented, and order-independent?	4	0	Stories;
Does the team compare its progress to the plan at predefined, timeboxed intervals, no longer than one month apart, and revise its plan accordingly?	4	0	Iteration;
Does the team make delivery commitments prior to each timeboxed interval, then nearly always deliver on those commitments?	4	0	'Done, Done'
After a line item in the plan is marked "complete," do team members later perform unexpected additional work, such as bug fixes or release polish, to finish it?	0	25	'Done, Done'
Does the team nearly always deliver on its release commitments?	3	0	Risk Management;

Collaborating

Question	Yes	No	Methodology Under Test
Do programmers ever make guesses rather than getting answers to questions?	0	75	The XP Team;
Are programmers usually able to start getting information (as opposed to sending a request and waiting for a response) as soon as they discover their need for it?	4	0	Sit Together;
Do team members generally communicate without confusion?	4	0	Sit Together; Common Language;
Do nearly all team members trust each other?	4	0	The XP Team; Sit Together;
Do team members generally know what other team members are working on?	1	0	Stand-Up Meetings;
Does the team demonstrate its progress to stakeholders at least once per month?	4	0	Iteration Demo; Reporting;
Does the team provide a working installation of its software for stakeholders to try at least once per month?	1	0	Iteration Demo;
Are all important stakeholders currently happy with the team's progress?	3	0	Iteration Demo; Reporting; Customer Involvement;
Do all important stakeholders currently trust the team's ability to deliver?	3	0	Trust; Reporting;

AYA 1.0.0

Developing

Question	Yes	No	Methodology Under Test
Are programmers nearly always confident that the code they've written recently does what they intended it to?	25	0	TDD;
Are all programmers comfortable making changes to the code?	25	0	TDD;
Do programmers have more than one debug session per week that exceeds 10 minutes?	0	3	TDD;
Do all programmers agree that the code is at least slightly better each week than it was the week before?	25	0	Refactoring;
Does the team deliver customer-valued stories every iteration?	3	0	Iterations;
Do unexpected design changes require difficult or costly changes to existing code?	0	3	Simple Design;
Do programmers use working code to give them information about technical problems?	1	0	Spikes;
Do any programmers optimize code without conducting performance tests first?	0	3	Performance Optimization;
Do programmers ever spend more than an hour optimizing code without customers' approval?	0	3	Performance Optimization;
Are on-site customers rarely surprised by the behaviour of the software at the end of an iteration?	4	0	Incremental Requirements;
Is there more than one bug per month in the business logic of completed stories?	0	3	Customer Tests;
Are any team members unsure about the quality of the software the team is producing?	0	1	Exploratory Testing;

AYA 1.0.0

Releasing

Question	Yes	No	Methodology Under Test
Can any programmer on the team currently build and test the software, and get an unambiguous success/fail result, using a single command?	25	0	Iterations;
Can any programmer on the team currently build a tested, deployable release using a single command?	5	0	Iterations;
Do all team members use version control for all project-related artefacts that aren't automatically generated?	25	0	Version Control;
Can any programmer build and test the software on any development workstation with nothing but a clean check-out from version control?	25	0	Version Control;
When a programmer gets the latest code, is he nearly always confident that it will build successfully and pass all its tests?	5	0	Continuous Integration;
Do all programmers integrate their work with the main body of code at least once per day?	4	0	Continuous Integration;
Does the integration build currently complete in fewer than 10 minutes?	4	0	Continuous Integration;
Do nearly all programmers share a joint aesthetic for the code?	1	0	Coding Standards;
Do programmers usually improve the code when they see opportunities, regardless of who originally wrote it?	4	0	Collective Code Ownership;
Are fewer than five bugs per month discovered in the team's finished work?	1	0	No Bugs;

AYA 1.0.0

Ideals

Question	Yes	No	Methodology Under Test
Do programmers critique all production code with at least one other programmer?	5	0	Pair Programming;
Do all team members consistently, thoughtfully, and rigorously apply all the practices that the team has agreed to use?	75	0	Pair Programming; Root Cause Analysis; Retrospectives;
Are team members generally focused and engaged at work?	5	0	Energized Work
Are nearly all team members aware of their progress toward meeting team goals?	7	0	Informative Workspace;
Do any problems recur more than once per quarter?	0	5	Root Cause Analysis;
Does the team improve its process in some way at least once per month?	5	0	Reviews;

12.3 New Employee Induction Form

ICNS 1.0.2

Induction Checklist for New Staff

We want to ensure you feel welcomed, fully equipped and settled in your new role and below is a list of information, tasks and activities which may help your induction process. The list is not exhaustive and there may be additional information required specific to the role, or faculty or service. We recommend that you go through each item with your manager in a meeting so that your manager can explain fully and give more context. Section 1 to 7 is for all staff and section 8 is for staff only. Items suggested for the first day is marked with .

Name:	Department / Service:
Line Manager:	Start Date:

TOPIC	STAGE			DISCUSSION AND COMMENTS
	First Week	First Month	Three Months	
1. WELCOME				
Introduction to line manager and team colleagues				
Company strategy				
Company structure, calendar, corporate policies				
Introduction to senior managers in your department or service				

Introduction to key colleagues within your department or service				
Introduction to key colleagues outside your department or service				
Review of your induction process				
2. YOUR ROLE AND LOCAL CONTEXT				
Job description and expectations of the role				
Probation – agree objectives and set the first meeting				
How your role contributes to the service and the company's strategy				
Diarise key meetings – one to ones, team and departmental meetings and other meetings required for the role				
Overview of the structure of your department including decision making process and "who does what"				
Process of planning and managing workload				
Communication channels within the company				

3. WORKING ENVIRONMENT				
Access to the building – key, ID card, code for doors (if applicable)				
Travel arrangements – car park, train, bus, car and cycle information				
Workstation and equipment required				
Reasonable adjustment required – support for disabled staff				
Photocopier and printing				
Telephone system				
Facilities – toilets, kitchen, shops and catering outlets				
Health and safety induction and online fire safety training (mandatory)				
Company building tour				

4. COMPANY SYSTEMS				
Email account and log-in password				
Staff intranet (news / events / staff directory)				
Wi-Fi connections				
Remote access				
Key IT policies (e.g. acceptable use, information security)				
Data protection				
Data security online training				
5. WORKING CONDITIONS				
Hours of work including arrangement for breaks.				
Payroll and pensions				

Process for reporting sickness and booking leave				
Process for claiming expenses				
Key HR policies (e.g. family friendly policies, sickness leave etc.)				
6. TRAINING AND DEVELOPMENT				
Training needs for your role				
Training opportunities for staff				
Career progression				
7. COMMUNITY AND SUPPORT				
Staff benefits (e.g. childcare voucher, healthcare scheme etc)				
Equality and diversity and online training (mandatory)				
Feel Good Health and wellbeing initiatives				

Sustainability				
Staff association (Staff Social) Various events organised throughout the year				
8. INFORMATION FOR STAFF				
Professional expectations (discussion with line manager)				
Research strategy and funding opportunities				
Workload allocation management system				
Information on key processes				

12.4 Google Java Style Guidelines

Google Java Style Guide

Table of Contents

1 Introduction

This document serves as the **complete** definition of Google's coding standards for source code in the Java™ Programming Language. A Java source file is described as being *in Google Style* if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving *advice* that isn't clearly enforceable (whether by human or tool).

1.1 Terminology notes

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (`@interface`).
2. The term *member* (of a class) is used inclusively to mean a nested class, field, method, *or constructor*; that is, all top-level contents of a class except initializers and comments.
3. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

1.2 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Google Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

2 Source file basics

2.1 File name

The source file name consists of the case-sensitive name of the top-level class it contains (of which there is exactly one), plus the `.java` extension.

2.2 File encoding: UTF-8

Source files are encoded in **UTF-8**.

2.3 Special characters

2.3.1 Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

2.3.2 Special escape sequences

For any character that has a special escape sequence (`\b` , `\t` , `\n` , `\f` , `\r` , `\"` , `\'` and `\\`), that sequence is used rather than the corresponding octal (e.g. `\012`) or Unicode (e.g. `\u000a`) escape.

2.3.3 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. `∞`) or the equivalent Unicode escape (e.g. `\u221e`) is used. The choice depends only on which makes the code **easier to read and understand**, although Unicode escapes outside string literals and comments are strongly discouraged.

> **Tip:** In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

Example	Discussion
`String unitAbbrev = "μs";`	Best: perfectly clear even without a comment.
`String unitAbbrev = "\u03bcs"; // "μs"`	Allowed, but there's no reason to do this.
`String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"`	Allowed, but awkward and prone to mistakes.
`String unitAbbrev = "\u03bcs";`	Poor: the reader has no idea what this is.
`return '\ufeff' + content; // byte order mark`	Good: use escapes for non-printable characters, and comment if necessary.

> **Tip:** Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

3 Source file structure

A source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

Exactly one blank line separates each section that is present.

3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

3.2 Package statement

The package statement is **not line-wrapped**. The column limit (Section 4.4, Column limit: 100) does not apply to package statements.

3.3 Import statements

3.3.1 No wildcard imports

Wildcard imports, static or otherwise, **are not used**.

3.3.2 No line-wrapping

Import statements are **not line-wrapped**. The column limit (Section 4.4, Column limit: 100) does not apply to import statements.

3.3.3 Ordering and spacing

Imports are ordered as follows:

1. All static imports in a single block.
2. All non-static imports in a single block.

If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.

Within each block the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order, since '.' sorts before ';'.)

3.3.4 No static import for classes

Static import is not used for static nested classes. They are imported with normal imports.

3.4 Class declaration

3.4.1 Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

3.4.2 Ordering of class contents

The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways.

What is important is that each class uses ***some* logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

3.4.2.1 Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no other code in between (not even private members).

4 Formatting

Terminology Note: *block-like construct* refers to the body of a class, method or constructor. Note that, by Section 4.8.3.1 on array initializers, any array initializer *may* optionally be treated as if it were a block-like construct.

4.1 Braces

4.1.1 Braces are used where optional

Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a method, constructor, or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.

Examples:

```
return () -> {
  while (condition()) {
    method();
  }
};

return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

A few exceptions for enum classes are given in Section 4.8.1, Enum classes.

4.1.3 Empty blocks: may be concise

An empty block or block-like construct may be in K & R style (as described in Section 4.1.2). Alternatively, it may be closed immediately after it is opened, with no characters or line break in between (`{}`), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: `if/else` or `try/catch/finally`).

Examples:

```
// This is acceptable
void doNothing() {}

// This is equally acceptable
void doNothingElse() {
}
```

```
// This is not acceptable: No concise empty blocks in a multi-block statement
try {
  doSomething();
} catch (Exception e) {}
```

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, Nonempty blocks: K & R Style.)

4.3 One statement per line

Each statement is followed by a line break.

4.4 Column limit: 100

Java code has a column limit of 100 characters. A "character" means any Unicode code point. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section 4.5, Line-wrapping.

> Each Unicode code point counts as one character, even if its display width is greater or less. For example, if using fullwidth characters, you may choose to wrap the line earlier than where this rule strictly requires.

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. `package` and `import` statements (see Sections 3.2 Package statement and 3.3 Import statements).
3. Command lines in a comment that may be cut-and-pasted into a shell.

4.5 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

> **Note:** While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit *may* be line-wrapped at the author's discretion.

> **Tip:** Extracting a method or local variable may solve the problem without the need to line-wrap.

4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
   - This also applies to the following "operator-like" symbols:
     - the dot separator (`.`)
     - the two colons of a method reference (`::`)
     - an ampersand in a type bound (`<T extends Foo & Bar>`)
     - a pipe in a catch block (`catch (FooException | BarException e)`).
2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
   - This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.

3. A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
4. A comma (`,`) stays attached to the token that precedes it.
5. A line is never broken adjacent to the arrow in a lambda, except that a break may come immediately after the arrow if the body of the lambda consists of a single unbraced expression. Examples:

```
MyLambda<String, Long, Object> lambda =
    (String label, Long value, Object obj) -> {
        ...
    };

Predicate<String> predicate = str ->
    longExpressionInvolving(str);
```

> **Note:** The primary goal for line wrapping is to have clear code, *not necessarily* code that fits in the smallest number of lines.

4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section 4.6.3 on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

4.6 Whitespace

4.6.1 Vertical Whitespace

A single blank line always appears:

1. *Between* consecutive members or initializers of a class: fields, constructors, methods, nested classes, static initializers, and instance initializers.
   - **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
   - **Exception:** Blank lines between enum constants are covered in Section 4.8.1.
2. As required by other sections of this document (such as Section 3, Source file structure, and Section 3.3, Import statements).

A single blank line may also appear anywhere it improves readability, for example between statements to organize the code into logical subsections. A blank line before the first member or initializer, or after the last member or initializer of the class, is neither encouraged nor discouraged.

Multiple consecutive blank lines are permitted, but never required (or encouraged).

4.6.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`(`) that follows it on that line
2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line
3. Before any open curly brace (`{`), with two exceptions:
   - `@SomeAnnotation({a, b})` (no space is used)
   - `String[][] x = {{"foo"}};` (no space is required between `{{`, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
   - the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`
   - the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`

- the colon (`:`) in an enhanced `for` ("foreach") statement
- the arrow in a lambda expression: `(String str) -> str.length()`

but not

- the two colons (`::`) of a method reference, which is written like `Object::toString`
- the dot separator (`.`), which is written like `object.toString()`

5. After `,:;` or the closing parenthesis (`)`) of a cast
6. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
7. Between the type and variable of a declaration: `List<String> list`
8. *Optional* just inside both braces of an array initializer
   - `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid
9. Between a type annotation and `[]` or `...`.

This rule is never interpreted as requiring or forbidding additional space at the start or end of a line; it addresses only *interior* space.

4.6.3 Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

> **Tip:** Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

4.7 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

4.8 Specific constructs

4.8.1 Enum classes

After each comma that follows an enum constant, a line break is optional. Additional blank lines (usually just one) are also allowed. This is one possibility:

```
private enum Answer {
  YES {
    @Override public String toString() {
      return "yes";
    }
  },

  NO,
```

```
  MAYBE
}
```

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section 4.8.3.1 on array initializers).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes *are classes*, all other rules for formatting classes apply.

4.8.2 Variable declarations

4.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

Exception: Multiple variable declarations are acceptable in the header of a `for` loop.

4.8.2.2 Declared when needed

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

4.8.3 Arrays

4.8.3.1 Array initializers: can be "block-like"

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
new int[] {           new int[] {
  0, 1, 2, 3            0,
}                       1,
                        2,
new int[] {             3,
  0, 1,               }
  2, 3
}                     new int[]
                          {0, 1, 2, 3}
```

4.8.3.2 No C-style array declarations

The square brackets form a part of the *type*, not the variable: `String[] args`, not `String args[]`.

4.8.4 Switch statements

Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements (or, for the *last* statement group, *zero* or more statements).

4.8.4.1 Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, there is a line break, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

4.8.4.2 Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group.

Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

Notice that no comment is needed after `case 1:`, only at the end of the statement group.

4.8.4.3 The `default` case is present

Each switch statement includes a `default` statement group, even if it contains no code.

Exception: A switch statement for an `enum` type *may* omit the `default` statement group, *if* it includes explicit cases covering *all* possible values of that type. This enables IDEs or other static analysis tools to issue a warning if any cases were missed.

4.8.5 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, Line-wrapping), so the indentation level is not increased. Example:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

Exception: A *single* parameterless annotation *may* instead appear together with the first line of the signature, for example:

```
@Override public int hashCode() { ... }
```

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line; for example:

```
@Partial @Mock DataLoader loader;
```

There are no specific rules for formatting annotations on parameters, local variables, or types.

4.8.6 Comments

This section addresses *implementation comments*. Javadoc is addressed separately in Section 7, Javadoc.

Any line break may be preceded by arbitrary whitespace followed by an implementation comment. Such a comment renders the line non-blank.

4.8.6.1 Block comment style

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/*
 * This is          // And so           /* Or you can
```

```
 * okay.          // is this.          * even do this. */
 */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

> **Tip:** When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.

4.8.7 Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

```
public protected private abstract default static final transient volatile synchronized native str
```

4.8.8 Numeric Literals

`long` -valued integer literals use an uppercase `L` suffix, never lowercase (to avoid confusion with the digit `1`). For example, `3000000000L` rather than `30000000001`.

5 Naming

5.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`.

In Google Style, special prefixes or suffixes are **not** used. For example, these names are not Google Style: `name_`, `mName`, `s_name` and `kName`.

5.2 Rules by identifier type

5.2.1 Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

5.2.2 Class names

Class names are written in UpperCamelCase.

Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

5.2.3 Method names

Method names are written in lowerCamelCase.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit *test* method names to separate logical components of the name, with *each* component written in lowerCamelCase. One typical pattern is `<methodUnderTest>_<state>`, for example `pop_emptyStack`. There is no One Correct Way to name test methods.

5.2.4 Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with each word separated from the next by a single underscore. But what *is* a constant, exactly?

Constants are static final fields whose contents are deeply immutable and whose methods have no detectable side effects. This includes primitives, Strings, immutable types, and immutable collections of immutable types. If any of the instance's observable state can change, it is not a constant. Merely *intending* to never mutate the object is not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
static final ImmutableMap<String, SomeMutableType> mutableValues =
    ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

These names are typically nouns or noun phrases.

5.2.5 Non-constant field names

Non-constant field names (static or otherwise) are written in lowerCamelCase.

These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

5.2.6 Parameter names

Parameter names are written in lowerCamelCase.

One-character parameter names in public methods should be avoided.

5.2.7 Local variable names

Local variable names are written in lowerCamelCase.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

5.2.8 Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`)
- A name in the form used for classes (see Section 5.2.2, Class names), followed by the capital letter `T` (examples: `RequestT`, `FooBarT`).

5.3 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following

(nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
   - *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
   - ... each word, to yield *upper camel case*, or
   - ... each word except the first, to yield *lower camel case*
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

Prose form	Correct	Incorrect
"XML HTTP request"	`XmlHttpRequest`	`XMLHTTPRequest`
"new customer ID"	`newCustomerId`	`newCustomerID`
"inner stopwatch"	`innerStopwatch`	`innerStopWatch`
"supports IPv6 on iOS?"	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
"YouTube importer"	`YouTubeImporter` `YoutubeImporter` *	

*Acceptable, but not recommended.

> **Note:** Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

6 Programming Practices

6.1 `@Override` : always used

A method is marked with the `@Override` annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

Exception: `@Override` may be omitted when the parent method is `@Deprecated` .

6.2 Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an `AssertionError` .)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment *if* its name is or begins with `expected`. The following is a very common idiom for ensuring that the code under test *does* throw an exception of the expected type, so a comment is unnecessary here.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

6.3 Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

6.4 Finalizers: not used

It is **extremely rare** to override `Object.finalize`.

> **Tip:** Don't do it. If you absolutely must, first read and understand *Effective Java* Item 7, "Avoid Finalizers," very carefully, and *then* don't do it.

7 Javadoc

7.1 Formatting

7.1.1 General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when the entirety of the Javadoc block (including comment markers) can fit on a single line. Note that this only applies when there are no block tags such as `@return`.

7.1.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (`*`)—appears between paragraphs, and before the group of block tags if present. Each paragraph but the first has `<p>` immediately before the first word, with no space after.

7.1.3 Block tags

Any of the standard "block tags" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When a block tag doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the `@`.

7.2 The summary fragment

Each Javadoc block begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with `A {@code Foo} is a...`, or `This method returns...`, nor does it form a complete imperative sentence like `Save the record.`. However, the fragment is capitalized and punctuated as if it were a complete sentence.

> **Tip:** A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to `/** Returns the customer ID. */`.

7.3 Where Javadoc is used

At the *minimum*, Javadoc is present for every `public` class, and every `public` or `protected` member of such a class, with a few exceptions noted below.

Additional Javadoc content may also be present, as explained in Section 7.3.4, Non-required Javadoc.

7.3.1 Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like `getFoo`, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".

> **Important:** it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named `getCanonicalName`, don't omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

7.3.2 Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.

7.3.4 Non-required Javadoc

Other classes and members have Javadoc *as needed or desired*.

Whenever an implementation comment would be used to define the overall purpose or behavior of a class or member, that comment is written as Javadoc instead (using `/**`).

Non-required Javadoc is not strictly required to follow the formatting rules of Sections 7.1.2, 7.1.3, and 7.2, though it is of course recommended.

12.5 Exploratory Testing Template

CUBIXEL

ETD 2.0.0

TutorPoint Integration & Testing Report - Iteration

Information

Testing Manager Name	Testing Manager Signature	Charter	Date of Testing
Daniel Bishop	Daniel Bishop	1.	##/##/2020
Eric Walker	Eric Walker		

Testing Results

Module Being Tested	Test Performed	Expected Outcome	Observed Outcome	Test Passed (Y/N)	Notes

12.6 Last One Board

CUBIXEL

LOB 1.0.0

Last One Board

Date:

Item Code	Item Name	Date of Re-Order

12.7 QA Metric Testing

QAMT 1.0.2

QA Metric Testing

Date:

Document Description

This document contains all QA Metrics of the managers within the company. It is the job of the QA Manager to ensure all managers are performing tests on and achieving their metrics. If this is not the case the QA Manager should discuss with the manager any issues preventing them from performing that task and develop a solution.

Project Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Work Efficiency	Total number of hours programming of completed user stories by the software development team divided by the number of user stories completed at the end of a single project iteration.		
Team Productivity	Average working hours of the whole software development team per day at the end of a week every Thursday.		
Team Contentment	Number of overall positive responses in individual feedback from each team member on the management and organisation of the project before each iteration.		
Reachable Project Scope	The overall percentage iteration completion of the assigned user stories at the end of the iteration period.		

Client Satisfaction	Percentage of number of positive responses against negative responses from the client / on site customer after each project iterations/revisions during the project's lifespan.		

Financial Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Percentage Iteration Progress Expenditure	Difference between percentages of expenditure and completion for each week every Thursday.		
Labour Buffer	Remaining surplus hours from the budget which can be dynamically assigned to employees for overtime. Tracked weekly every Thursday.		
Development Cost Difference	Difference between budgeted programming cost against actual iteration cost. Tracked fortnightly every Thursday.		

CUBIXEL

QAMT 1.0.2

Marketing Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Product Awareness - Profile Visits	The total number of people that have visited/ viewed the product social media account over the period of a week.		
Product Awareness - Profile Reach	The total number of people that have viewed/ interacted with any of the posts made by the product social media account over the period of a week.		
Product Growth	The total number of people following the product social media account at the time of measurement.		
Product Demand	Percentage of surveyed people currently using or looking for the product or similar products, measured at the beginning and end of the development period.		

QAMT 1.0.2

Technical Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Software Development Team Agility	Using the 'Assess Your Agility' self-assessment quiz. This should be done at the beginning of each iteration to review the previous iterations performance. Lowest Value Recorded.		
Time Estimates for User Stories	Difference between estimated time and actual time for completion of user stories so as to refine estimates. This should be done at the end of each iteration as a sum of all stories. Further time estimates should then be updated based on the insight from this check.		
Non-Compliance for Coding Standards	A code review should be done for each story completed and marked as 'checked' alongside the user story card. Any non-compliances should be highlighted and recorded. The number of non-compliances should be reduced as the team learns the coding standards. Code should also be reviewed for adherence to standards imposed by outside regulatory bodies as and when required.		

CUBIXEL

QAMT 1.0.2

Design and Specifications Manager

Metric	Measurement	Being Measured (Y/N)	Notes
User Experience Satisfaction	Difference in number of positive responses against negative responses gained from focus groups and user interviews on using the product directly. This is measured at the end of each iteration, starting from the 2nd iteration.		
Similarity to Design Vision	Difference in number of positive responses against negative responses from feedback and discussions with the client on the product design philosophy and specification. This is measured at the start of every Thursday, starting from the 2nd iteration.		
Complementation to Project Vision	Number of redesign iterations initialised by the client or on site customer, including removed design features. This is measured continually throughout iterations with a total number of redesigns for each work week every Thursday.		

CUBIXEL

QAMT 1.0.2

Quality Assurance Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Coherence to Management QA Metrics	Use the assess the 'QA Metric Testing' document and check that all QA Metrics have a test and are being satisfied. This should be done at the end of every iteration, measured as a percentage of metrics being currently assessed.		
Employee Comprehension of Company Ethos	Complete random audits once every four weeks on one member of the team using the 'Team Checker' document. Number of non-compliances noted and raised if necessary.		
Document Standardisation	Company documents proofread every other iteration with any issues noted and raised. Number of non-compliances recorded and highlighted on the 'Document Checker' document.		

CUBIXEL

QAMT 1.0.2

Testing and Integration Manager

Metric	Measurement	Being Measured (Y/N)	Notes
Quality of Code	Number of bugs found in code in a working week via exploratory testing of the simulated program before code is subject to testing.		
Quality of Tests	Expected test outcome against the actual test outcome, record the proportion that arises due to insufficient module tests via exploratory testing for a working week.		
User Story Integration Latency	Measure and record the length of time between each user story being submitted for 'done' exploratory testing and being integrated into the development repository each iteration.		