In my process of developing the core entities, I drew heavily from the TM354 Software Engineering module materials. I documented every noun that could potentially represent an application object, which already made me question the completeness of my initial understanding. I then reviewed existing paper inspection forms thoroughly, trying to shadow workflows and capture all relevant terminology. This exercise led me to compile a spreadsheet of fields and a list of domain terms, some of which matched my initial conceptualisations but also revealed gaps.

I realised early that entities like "Inspection," "Engineer," "Manager," "Checklist," "Inspection result," "Item," and "Asset" could form the backbone of a conceptual class diagram, but I questioned whether these were sufficient or too simplistic for the complexity of real-world scenarios. The creation of some core entities was a mix of brainstorming and analysing paper documents, which I knew wasn't perfect, but was the best approach I had at the time.

I identified recurring fields such as Date, Location, and Sub-checks across different forms, an insight that seemed promising at first. Recognising these commonalities felt like a step toward better database design, but I wondered if this generalisation might oversimplify intricate workflows. I discussed this with the senior technical advisor, who assured me that this would make the application more flexible and less tied to specific forms, yet I couldn't shake the suspicion that this approach might miss nuances.

From these forms, I generated a glossary and outlined key processes, such as the engineer’s completion and the manager’s review, and documented business rules. These rules, such as requiring comments on failed items or passing inspections only if all mandatory sub-checks are successful, added layers of complexity that I had to consider carefully when translating them into schema constraints.

Next, I converted repeated fields into conceptual classes and attributes, such as User, Site, Zone, and Inspection. I aimed to keep methods and implementation separate to avoid early binding decisions. This step resulted in a classes and attributes diagram using Visual Paradigm, which I hoped would serve as a solid foundation. However, I worried it might still be too abstract or disconnected from the actual workflows.

Following my tutor's guidance, I modelled the workflows of both Engineers and Managers to see how entities interact over time. This process uncovered additional requirements, such as attachments and comments for failed inspections, and confirmed that only one active inspection should occur at a time. These constraints seemed logical, but I was aware that real-world scenarios might challenge these assumptions. Validating these with stakeholders was reassuring, yet I questioned whether the model truly captured all edge cases.

Overall, this process has been a balancing act between structured analysis and the recognition that real-world complexities often defy neat categorisation. I remain critical of my assumptions, but hopeful that these foundational steps will support a flexible, scalable, and realistic application. diagrams.

As a next step, I mapped out the requirements to understand the application's data needs, focusing on user requirements such as monthly emergency light tests and safety checks. I realised I would need to create reusable checklists, optional numeric readings (to support meter readings), and the possibility of additional attachments, such as photos. This is so I wouldn't have to rely on hard-coded forms. This approach helped create a flexible structure that supports various inspection types without requiring code changes.

I split the checklist into **templates** and **instances** so that we can modify checklists over time without breaking existing data or rewriting code. Templates are the definitions we author (INSPECTION\_TEMPLATES, SUBCHECK\_TEMPLATES). When an engineer starts an inspection, the app **clones** those sub-checks into INSPECTION\_SUBCHECKS and stores the key details with the copy (name, value type, pass rule, mandatory).

**Why?** We need to edit checklists as the process evolves, but we also need past inspections to stay exactly as they were.

**So what?** Managers can publish updates without code changes, engineers always see a clean, consistent form, auditors get stable records, and repeating an inspection just makes a new instance from the latest template. Results are saved against the instance rows (not the authoring tables), and the overall pass/fail is calculated from the mandatory items. This keeps the data simple, **quarriable**, and easy to maintain.

I translated the earlier-formed class diagrams into a proper database schema, defining relationships and validating them with the process models. I made sure to specify cardinalities, such as users inspecting multiple inspections and inspections occurring at different sites and zones. This locks in the relationships before choosing specific keys.

I then selected primary and foreign keys based on standard database design principles, ensuring they are unique and minimal. I chose alternate integer keys for frequently referenced tables and used natural keys where appropriate, like usernames and site IDs. These choices follow best practices to ensure straightforward and reliable data joins.

Next, I took the attributes created as part of the classes and attributes document and extended it by adding cached fields in instances to preserve history. For templates, I included attributes like category, check type, name, and version. And for the instances, I cloned relevant data into cached fields, timestamps, and attachment paths, ensuring we keep an immutable record of each inspection.

I applied normalisation techniques from 1NF up to 3NF, removing repeating groups, ensuring dependencies are correct, and moving multi-valued data into separate tables. This keeps our database clean, reduces anomalies, and ensures data integrity.

I created views for derived data, such as overall pass/fail status, rather than storing constants. These views automatically aggregate sub-check outcomes, ensuring our data remains consistent and synchronised without manual updates.

I added constraints (like enumerations and unique keys), indexes for common queries, and versioning controls to manage different template versions. This ensures our database enforces business rules and performs efficiently.

I implemented privacy and data retention measures in line with GDPR, keeping only necessary personal data, setting retention policies for attachments, and considering pseudonymization for exports. These steps protect user data and comply with legal requirements.

Finally, I translated the ERD into SQL statements, creating tables and constraints, and seeded some example templates, such as the Emergency Lights Monthly check, so we could test the entire process from start to finish.