Team 11 - BicyclERP **E**ngineer **R**etrospective **P**ostmortem

Introduction

Enterprise resource planning systems, or simply ERP systems, are software solutions that aim to automate business processes, drawing data from multiple departments within an organization. Often customizable to fit the enterprises' needs, an ERP system helps integrate, plan and manage these important processes all in one place. Inside such a system, data flows between interwoven modules such as the inventory, supply chain, manufacturing, procurement, accounting and more to improve efficiency, save costs and respond to consumer needs and market demands.

In our case, the goal of the project was to develop an ERP solution for a bicycle manufacturing company. To do so, we've considered and implemented features in the following areas: planning, scheduling, vendor relations, production, quality management, packaging, shipping, sales and accounting. The end users of the solution can be categorized into two distinct types: the administrators (internal to the company) and clients (external customers). An administrator manages the creation of products, the procurement of materials from vendors, the inventory, the orders, processes sales, and may access all the internal information needed to monitor the health of the business. On the other hand, a client has access to a different public interface, which permits them to put in orders, obtain confirmation information and follow up on their purchases.

Coming into the project, we had mixed expectations. At the beginning, there was a feeling of uncertainty as many did not fully understand what an ERP solution was. However, with most members of the team having had experience working on social media platforms (i.e. in web development) from previous courses, coming to an agreement of what tech stack to use to tackle the problem was reasonably easy. Of course, each one of us expected that the choosing of the frameworks would be in accordance with what the majority was comfortable with, while the rest may have to learn them on the fly.

In the end, our project was a reasonable success. While the front-end may look unpolished and clunky for the time being, the functional requirements were satisfied on the back end side.
The final result was a working web application that includes the complete set of features that were planned ahead and expected from our client. While there is certainly room for improvement, we are proud that the system met all the requirements that the product owner communicated with us.

The application was built using React for the front end, Spring Boot for the back end and follows the Model View Controller (MVC) architecture. All data were stored in a MySQL database and Flyway was our go-to migration tool. We made use of gradle to build the project, and a docker

container to package and run it. This way, all the parts needed by our project, including libraries and other dependencies were contained in one place, virtually, separate from our local environment. To ensure the stability of our program code, circleCI enabled our continuous integration approach, and JUnit was adopted as our testing framework. In addition, we ran Codecov to ensure a reasonable code coverage ratio. In terms of planning and version control, Zenhub and GitHub were picked. Finally, Figma was our design tool to create wireframes for each sprint.

What went wrong

1 - Task Allocation and Management

On multiple occasions, tasks that were available did not have sufficient information for one to pick up and start. They needed to talk to someone in order to fully understand the scope of that task. That is, certain tasks overlapped one another, making it difficult to get started. In addition, tasks were oftentimes not broken down to elementary components to be digestible by a single person (i.e. the person working on the task).

Task allocation, management and understanding issues occurred especially at the beginning of the project, and at the beginning of each sprint. This is because those are the moments when the features to implement are the least understood. This caused delays and frustrations among the members.

To address this problem, open communication and meetings with our product manager was the best way to clarify the functional and non-functional requirements, as well as the content needed in our documentation.

To avoid repeating this kind of mistake, one or two people on the team should be specifically put in charge of breaking down the tasks more carefully by writing more detailed descriptions and clarifying assumptions early on with the product manager.

2 - Catch Expressions

On multiple instances of the project, there are occasions where developers had used a try-catch statement where the exception class was Exception which, while can be used, catches all exceptions without remorse. This can and will lead to issues down the line as some exceptions that should have crashed the system do not, leading to incorrect information being manipulated.

```
}
catch (Exception e){
    return new ResponseEntity<>("material of the id not exist", HttpStatus.CONFLICT);
}
```

During development, some members have chosen to use the base exception class as a one-size-fits-all option for all exceptions. While it should never be used, it still does what the developer had intended it to do. However, precaution must be taken and impacts must be weighed for such decisions.

Overall, there was no impact in terms of testing. However, as the lines of code multiplied and the system started getting larger, some more specific exceptions that were meant to be caught may cause unpredictable behaviors in the system if left unattended.

This issue was not addressed as it was found late in the project, and thus nothing was done to rectify it.

The best practice to prevent this is to find the exact exception that was meant to be caught, and substitute the general exception class for it. In retrospect, deciding on stricter coding conventions should be discussed and established beforehand with the team.

3 - Docker & Database

Due to existing configurations of various softwares on the each members' local environment, some technical conflicts arose with the installations. Most, if not all members needed to carefully edit their environment in order to access the database used in our project. As an example, for MySQL to be accessible via our docker container, previous and local MySQL services must be shut down. Although seemingly a simple issue, it took us quite some time to figure out the root cause of our inability to run the database.

This problem appeared in sprint 1 and 2, and caused a slow down in our development speed from the get-go, and hindered our ability to estimate our effort.

The issue was addressed mainly by troubleshooting efforts, Googling and meeting with the one who wrote the docker files. Miscommunication and a lack thereof was also a reason this caused more delay than it should have.

Thinking back, this could have been done better if the individual cases and troubleshooting efforts were documented into the README rather than left as messages on our Discord channel (which were often overlooked at first).

4 - Coding Conventions

At the start of the project, the team did not have an agreed upon coding convention to follow. Since each member came from different technical backgrounds, the code was written in different styles with minimal presence of comments or logs explaining what it does.

All things considered, there was minimal negative impact on the project as those who worked on related features tended to program together and maintain active communication. As long as the written functions were explained, others could simply plug and play.

This problem was addressed when the team had a meeting about enforcing comments, logs, and a standard naming convention.

In hindsight, code clarity should have been something that was taken more seriously from the start of the project.

What went right

1 - Task Planning

From the beginning, user stories and the UI prototyping were made before the start of each sprint. This allowed each individual to work start and work at their own pace. Having had a blueprint outlining the construct of our project-to-be also helped give direction for each sprint.

In addition, due to meticulous planning, no surprise large tasks were added while a sprint was in progress. Also, at the end of each sprint, all tasks' status' were known and sorted using Zenhub.

Good and active planning made it easy to know what issues may have needed to be cancelled, moved, delayed, or re-assigned. Having a clear view of who does what made it easy to know who to directly address when an issue has not been updated.

2 - Time Management

All tasks and documents were done on time and very rarely had an issue been delayed and pushed to a subsequent sprint. Also, no major feature originally planned had to be cancelled. Overall, there were no surprises in terms of tasks and their expected deadline. Thus, during the last day of a sprint, all tasks had their status properly changed to completed, cancelled, or delayed.

3 - Communication and Support

Communication within the team was effective. We have maintained open communication 24/7 on Discord, and called for impromptu meetings here and there to clarify assumptions, assign tasks and steer ourselves in the right direction. Also, if one developer had issues with their tasks, there was always someone who could provide assistance. Since tasks overlapped with each other, many had opted to program in pairs to efficiently workout their solution. This led to little instances of merge conflicts as well as no overhaul of the existing code.

Conclusion

In conclusion, the main aspect that every member learned was how to communicate and work with other members within a project team. Furthermore, it gave us a glimpse of what would comprise an ERP system. Our team gained valuable experience on the front-end, back-end, and documentation side of web development.

The main takeaway for this project is that code development is perhaps one of the least important aspects in a large project. First and foremost, the project team should be concerned with the design of such a large scale system. It should consider all the eventual and potential changes that could happen in the user requirements. Secondly, what really becomes important in the long run is the documentation that was left behind. This includes the wireframe, javadocs, and supporting documents about the project's goal, architecture, and testing.