Postmortem Analysis

Double Black Diamond ERP

TEAM12

Tyler Znoj 40005987
Killian Kelly 40014508
Abdul Sirawan 40074202
Adrien Kamran 40095393
Qandeel Arshad 40041524
Cedrik Dubois 40064475
Anthony Chraim 40091014
Matthew Giancola 40019131

A report submitted in partial fulfilment of the requirements of SOEN 390 Concordia University

Date 20/04/2021

1 - Introduction

1.1 - In what context did we build this software?

This software was built as part of an 8-person team project for the Concordia Software Engineering mini-Capstone.

1.2 - What were our instructions?

Our end goal was to develop and release a software product according to a set of requirements given by a product owner. In our case, the product was to be an ERP (Enterprise Resource Planning) service application for the manufacturing, sale, and management of bicycles. Our product's requirements were provided to us by Fatima Sabir, our simulated product owner and our actual TA.

2 - Methodology

2.1 - What project management system did we use?

We used the Agile project management system. We used Scrum, which utilizes the Agile mindset and worked in iterations, or a series of sprints, which are short periods of time in which the team works to complete a set amount of work.

In JIRA, we can create a sprint and fill it with issues from the backlog. We do this by simply dragging and dropping the issues into the sprint. Once the issues are added to the sprint and the team is ready, we click on "Start Sprint." We can also select start and end dates for that particular sprint. With the sprint started, we can monitor the team's progress by moving issues from To Do to In Progress and Done.

Once all the issues are added to Done, we can close the sprint by navigating to Active Sprints and pressing the "Complete Sprint" button.

2.2 - How did we deal with JIRA issues?

Our team decided to use a website called scrumpoker.online to attribute story points. In this website, a session was created and the team members were added. The user stories were written onto cards, and once the session began, the team members would vote a number of story points for each of the cards corresponding to a user story. The story point with the most votes would be chosen for that particular user story. However, story points that varied from the majority vote would be reviewed, and the members that voted for it would be asked why they chose that number in order to look at different points of views. A consensus would be drawn and only after that would the story points of each user story be chosen.

Once the story points were attributed, the team would split the tasks, each member choosing the task they want to work on. Both attributing story points and splitting tasks would often take place in the same team meeting.

2.3 - How did we structure our releases?

Our sprints were mostly around 20 days long, with the exception of Sprint 1 which was 21 days and Sprint 5 which was 12 days. Our deliverables consisted of the code deliverables for that particular sprint, as well as the documents. The documents were updated with each passing sprint.

All sprints included:

- The Software Architecture Description: Contained information about our project, architecture evaluations, rationale for key decisions, stakeholders and concerns, different viewpoints, and diagrams for views corresponding to the viewpoints including the class diagram.
- The Testing Plan: The testing procedure and test coverages for the sprints.
- The Release Plan: A thorough plan in the form of a spreadsheet for each sprint as well as the progress for every task. The start and end dates for sprints, their durations, status, comments, JIRA issue tags as well as Github links were all included.
- The Risk Management Plan and the Risk Log: Included the Risk Management procedure, Risk Matrix, Risk Response Planning, Tools and practices and Risk Management Plan Approval.
- UI prototypes: The prototypes/mock-ups for the user stories of that sprint.
  Sprints 2 and 3 additionally included:
- Defect Tracking Report/Bug Report: Consisted of different bugs encountered, their error messages, status, reporters, resolvers and history.
  Sprint 4 did not include a Bug Report and instead included:
- Code Quality Report: The methodology including what in our code base we analyzed, how we analyzed it, how we parsed it, the results, its analysis, and the conclusion.

2.4 - How did we achieve parallel development?

The entire project was on our GitHub repository. Each sprint had its own branch from which the features' branches would split off of. Each feature was developed on its own branch to isolate changes and minimize conflicts. Once a feature was finished, it would be merged with the sprint branch after being reviewed by team members who did not work on the feature. By having non-involved members review the requests, they would review the changes with a neutral point of view as they did not develop the function. Once a sprint was done, the sprint branch would be merged with the master branch and would have a similar review process.

2.5 - How did we plan for the long-term development of the software?

All of the documents had their first version written during the first sprint. After which, they were updated each sprint. By having all of the documents written at the beginning of the project, our team was aware of what was needed to complete the project and what issues we should be aware of. This allowed us to plan the order of the features developed and what non-functional requirements would be needed to be addressed early in the project. By consistently updating our design documents, we were able to readjust our plans for future releases and be reminded of our direction.

3 - Technology

3.1 - Which frameworks/libraries did we pick and why?

- Django: the core of our web application. Provided the file structures and underlying Python libraries for templates, views, and data models. Also came packaged with authentication checks (encryption/hashing for security) built-in, alongside a wide variety of other useful features.
- Bootstrap: the front-end framework for our web application. Mostly provided a unified style for our front-end elements, and allowed for mobile viewing.
- Psycopg2: this is how we linked our Django setup with our external PostgreSQL database.
- ReportLab: used to export our data tables from the application into CSV files for offline viewing.
- Mixer: used to generate dummy data in our PostgreSQL database for testing purposes.
- Coverage: used to measure the code coverage of our test cases.
- Django-Mathfilters: allowed for the use of subtraction, multiplication and division in Django template tags (addition being default).
- Pylint: used to measure the quality of our project's code according to a number of rules and categories. Results from this static analysis were used in our Code Quality report.
- Gunicorn: used to implement an HTTP server for our Docker container, and then eventually for our Heroku deployment.
- WhiteNoise: used to serve static files directly from within the application instead of a CDN. Implemented when deployed with Heroku.

3.2 - Which other software tools did we use, and how were they integrated?

- Docker: used to containerize our entire application and make it platform-agnostic. Integrated by having the requirements for the project included as a text file, then using Docker Desktop to run a Docker Compose file from within the project to generate an image file.
- Heroku: used to deploy our application on the web. Heroku works like github, where you push your code to a heroku repository. By pushing the code, heroku makes sure all the requirements are installed in the production environment.
- Bootstrap Studio: used to generate static Bootstrap-compliant assets and templates for use in the project, and for prototyping user interfaces. Once designed in-app, Bootstrap

Studio could export the necessary HTML, CSS and JS files needed to display the template in Django after minimal tweaking.

4 - Features

4.1 - List of features and explanation

ERP-14: As an administrator, my system should only allow registered users to access information to protect confidential information

Implementation: This was done by using Django's @login_required decorator. This decorator checks if the user is registered, if so it renders the view. If the user is not authenticated, it redirects the user to the login page.

ERP-15: As a user, I want the system to automatically generate logs and audit trails so that I can see when and where issues happen

Implementation: This was done by using Python's Logging method. It allows for developers to print to console and to a file when the function is called and can be customized to include things such as the date and time.

ERP-16: As a user, I want to generate documents in different formats to eventually review or share them

Implementation: This was implemented with the Python library "ReportLab". The data tables in each module's "history" tab were loaded into ReportLab's buffer, then exported in .CSV format for offline viewing.

ERP-17: As a user, I want the system to link to some Cloud storage service so that my data is backed up

Implementation: This was completed by creating a button and linking it to the URL for google drive. If the user is already logged in to their drive, they can access it from our system, or else they are asked for credentials to log in.

ERP-21: As a user, I want the the system to send e-mails so that I can notify my co-workers

Implementation: This was completed by creating a notification system which logs any changes to the system and displays them in the bell icon. Also, whenever a new account was created, the system would send an email to the provided email address welcoming them. The notification system only examined changes in the customer model. It could have been extended to other models.

ERP-19: As an administrator, I want my system to support automatic logout to protect confidential information and to reduce server load

Implementation: This is as simple as setting 2 variables in the settings.py file. The first one is the timer where after 30min, logs the user out. The second specifies to refresh the timer after a request has been issued.

ERP-11: As a user, I want to be able to track any and all costs

Implementation: This was accomplished using data querying and frontend display. After any action that incurred cost or made profit, a transaction record is created with the information and it is then viewable in the Accounting module.

ERP-13: As a user, I want to track quality data in order to identify/remove defective products and increase the overall product quality

Implementation: For this feature modifications to the backend were needed to create unique records for every raw material/part/product. This made it possible to mark them as functional or defective in the Inventory module. If a part is marked as defective, it is ignored for production but remains in the inventory until “fixed” or removed.

ERP-20: As a user, I want to be able to use this system on my mobile device

Implementation: Bootstrap is an inherently mobile friendly frontend framework. The template that we used was initially designed to also be compatible with mobile screens.

ERP-22: As a user, I want the system to support process driven work-flow messaging to increase my team's productivity

Implementation: Refer to the implementation of ERP-21.

ERP-8: As a user, I want to create, edit, and track material lists

Implementation: Material lists are a combination of records in the MadeOf table. A product can be made of several parts including raw material, therefore these relationships are stored in the MadeOf table, and to query the material list of a specific part/product, we filter this table for the parent part. This returns the list of parts that make up the part/product. It also allows us to have a chain of relationships where each sub part can be made of other sub parts.

ERP-10: As a user, I want to define, order, and track raw material and final products

Implementation: A raw material is stored in the Part table as type “Raw Material”. A sub part is type “Part”, and product is type “Product”. This lets us differentiate between the 3 types. All 3 are kept in warehouses in the Contain table. In the contain table, a record for each individual

raw material/part/product is created but they all link to a record in the Part table. In other words, when we define a raw material/part/product, the record is created in the Part table. When we order individual records are created in the Contain table. We then query the Contain table to track everything.

ERP-9: As a user, I want to track inventory

Implementation: Refer to the implementation for ERP-10.

ERP-31: As a user, I want to be presented with different user interfaces depending on my access level

Implementation: This was done by using Django's "Groups" functionality where we can assign each user certain groups and privileges. In the base template, for every module we have a check and if the user is not assigned to the required group, the respective button does not show up. This check is also done when rendering the views, where the user needs to be part of the group to be able to render the view. With this, we are able to present different interfaces depending on the groups the user is associated with.

ERP-34: As a user, I want to track the delivery/shipping status of products I sell

Implementation: A table was constructed containing all placed orders. Inside that table, there's an order status where it displays the status of the order whether it is: Shipped, Pending, Delivered.

ERP-35: As a user, I want to edit an existing material list

Implementation: This was achieved by loading an existing material list for the part/product that needs to be updated. The system will prompt the user to add or remove the existing raw materials. Then, when all changes are done, the user can hit the create material list button and it will be updated in the back-end.

ERP-53: As a user, I want to be able to create vendors and manage which vendors are currently selling which raw material

Implementation: This was completed by having a form that registers new vendors and adds them to the system. Then a table was created in a new tab where it displays the inventory of a selected vendor. The table prompts the user to remove an entire batch of raw materials. Also, there is another tab where it allows the user to create/edit a raw material for a selected vendor.

ERP-54: As a manager, I want to be able to see the list of my employees and their privileges in the system

Implementation: Managing the users' privileges in the system is designed for superusers only. A super user can click on his/her name in the top-right corner, a drop down will pop up with Settings and logout options. Staff users will only see the logout option. The super user can then click on the settings and select a user from a list of users and decide on which group this user should belong to. That selected user will only be able to view the models that the super user allowed him/her to see.

ERP-55: As a user, I would like to export the histories of each module

Implementation: This was implemented by fetching the models that are used to display the information in the tables. Then, an http response is created and headers are added to it along with the information in the table. Then, when a user views a certain table which contains the history of that module, a download button is displayed at the top of the table allowing users to export the history in an excel format.

ERP-57: As a user, I would like to have a comprehensive overview of the modules from the home page

Implementation: This was implemented by creating 3 cards displayed at the top of the dashboard containing information about the profit/expense the company has. A bar chart is created displaying the numbers of raw materials orders, manufactured items, and sold products. In addition, a doughnut chart is used to display the top three sold products to give an insight to the user about their most popular products.

ERP-58: As a user, I would like to see my history of manufactured products/batches

Implementation: This was simply implemented by creating a table and populating it with the data from the Manufacture model. The manufacture model contains all the records of the manufactured items.

ERP-60: As a user, I would like the workflow messaging system to notify me of changes

Implementation: Refer to the implementation for ERP-21.

4.2 - Which features were planned for development, but ultimately fell through?

ERP-12: As a user, I want to connect to production machinery via different communication modes as to centralize the systems

This feature was suggested at the beginning of the project. Our team did not have a strong understanding of the requirements for the project. After discussing the requirements with the TA, we realized that there are more important features that need to be implemented. This feature was not requested by the requirement of the project. So, it was left out and we never pursued developing it.

ERP-59: As a user, I would like to create a portal for customers to order products directly

This feature was placed at a low-priority. It was suggested to be implemented in sprint 4. However, sprint 4 objectives were to test the code and refactoring and making sure that all functionalities work. However, we were ambitious to create more features which we succeeded in tackling. Those features were placed at a higher priority. Moreover, this feature suffered from shortage of manpower and it was never completed. So it was decided that it should be left out and focus more on making the core functionalities of the app more robust.

5 - Quality Assurance

5.1 - How did we track bugs?

Bugs were logged into a Bug Report document where the causes and effects of each bug were listed. If anyone encountered a bug, they would list it in the document and write the required details (what went wrong, what they tried to do, etc.) , provide screenshots and list any additional relevant information.

5.2 - How did we track risks?

Before any risks were tracked, a Risk Management Plan was developed by the team, in which we decided to track risks through the use of an excel file for a Risk Log and a document that contained the top ten risks. Whenever a risk was encountered or thought of, the person would enter it into the log and write the relevant details (the date, what the risk was, potential solutions, who will manage the risk, etc.). Each sprint, the top ten risks would be reevaluated and updated.

5.3 - How did we deal with bugs and risks?

Bugs and risks were dealt with differently. Bugs would either be dealt with immediately if possible or at the end of the sprint where they would all be evaluated as a batch. The team would often pair program to try to fix these issues.
Risks were often dealt with incidentally as the proper implementation of features would eliminate the potential risks. These cases would be updated at the end of the sprint when all the documentation was overlooked. In fact, most of the potential risks did not even occur.

6 - Results

6.1 - What went wrong?

6.1.1 - Database Migrations

Initially, we had a team meeting discussing the necessary models for our project. This meeting happened at the first sprint. Based on this early meeting, we developed an ERD displaying the models, their attributes, and their relationships. Later on, the models were created by a team of 3-4. We imported those models, and we added a command that migrates the created models when running ‘docker-compose up’.

However, in later sprints, we realized that we need to add more models and modify existing ones. That is when each team member started struggling in having a consistent migrations file. So, whenever a team member runs ‘docker-compose up’ it will fire up the command that migrates the inconsistent migration files. Which led to our database becoming inconsistent.

After documenting this bug and reporting it, we have researched the issue and found a fix where we have to delete all the migrations file on each end-user and re-doing the commands to create the migrations file and migrating the models.

Also, as our project started growing in each sprint, we were creating an app for each module which contained models specific to this app. So, when we create the migration files for the apps, we would have to migrate them one by one, to ensure consistency and that nothing is left out.

Eventually, when we deployed our project, we also deployed our database along with it. So the database for the project became consistent across all users because it is in the cloud, rather than having it on the local system for each developer.

6.1.2 - The Email System

At the initial stages of the development, we were interested in implementing a work-flow messaging system that sends emails to users. We were hoping that when a user registers their account, the system will send this user a message welcoming him/her. However, in order to achieve that, we used Mailtrap service to automate sending dummy emails, but not directly to the user but rather to the mailtrap inbox system where developers have access to it.

Nonetheless, to achieve that, we needed to access the Mailtrap service and use private credentials and add it to our settings file. This step was necessary to be able to register users without having errors. However, those credentials must be deleted from the settings before making a push to the github repository. Because they are private credentials and could be exploited and used maliciously.

Moreover, this was not an ideal way to achieve a work-flow messaging system. Thus, we researched other services that automate sending emails, and we came across SendGrid. SendGrid is an email marketing company that is easy to use. Integrating SendGrid with our app was relatively quick and easy. We did not have to configure our settings file anymore, and we

eliminated the risk of having our credentials exposed to the public. SendGrid allowed us to send emails to users upon their registration with the app efficiently.

6.2 - What went right?

6.2.1 - Pair Programming

Pair programming is an incredible way to ensure consistency among other team members. It is also beneficial to improve coordination between team members. In case of our project, we would schedule a team meeting where we discuss the user stories to be implemented for a given sprint. Then, we would split the tasks giving each task for two members at least. If we are dealing with an epic, or a user story that has many features to be implemented. We would increase the manpower on this epic.

Our team would volunteer to work on user stories rather than having to assign those user stories to members. This helped because each member would select the task that they are most comfortable with. This increased the productivity and motivation of team members.

Whenever an assigned team decides to work on a user story. We would have a meeting designated for that user story, we always start discussing the layout aspects in which we want to achieve this task. Then, the front-end team will generate the necessary templates and provide it to the back-end team. The back-end team will meet to discuss different approaches to achieve a certain feature and they will select the most optimal solution.

Then, one team member would share their screen and start writing the code. Whenever this person encounters a bug or an issue with the code, the spectators would research this issue and provide links/screenshots on how to resolve it. This method ensured understanding of the implementation of the code across all team members.

6.2.2 - Frequent team meetings

Team meetings were frequent and extensive. We would have a meeting at the beginning of the sprint to discuss the work that needs to be completed for a given sprint. After splitting the tasks, each sub-team will meet multiple times to work on the assigned task. Also, most of our team members were available in case one of us encountered a bug. Whoever is available will join a meeting and address the encountered bug and research a solution for it. The bug is then documented in the bug report for whoever missed that meeting can consult this document to resolve this bug in case it happens to them.

Before the end of a sprint, our team meets again to demonstrate what each sub-team has accomplished and how they accomplished it. This ensured that everyone is on the same page. A final meeting takes place before performing the demo of a given sprint to the TA. This meeting

compromises of integrating all features and testing them one last time before demonstrating them to the TA.

6.2.3 - Jira

Our team used the Jira software tool to track our development. Jira had incredible features where we would be able to list our targeted user story or tasks for a given sprint. We were able to add descriptions which further explains this user story. We also added any necessary links that could help in achieving this user story. We assigned team members that will target this user story. And we were able to add story points and select the level of priority for this given user story.

Once we add all the user stories or tasks for a given sprint, we would start this sprint and whenever a sub-team starts a user story they would mark it in progress, and when it is finished, it is marked completed. Using Jira helped the team in staying well coordinated, and it is a very efficient tool to remain organized during the development project.

6.2.4 - Backlog

The Backlog was the list of all the functional requirements. This list was derived from the mandatory requirements given to us by the professor, but all the requirements were rewritten as to be easily digestible and larger features were split into smaller issues. As the whole team was involved in this process, everyone was aware of what was needed.

This backlog was used in tandem with Jira and was created at the beginning of the first sprint. By having our backlog, the team was able to plan out all the sprints quickly and smoothly. Without the backlog, it would have been impossible to future proof our features.

6.2.5 - Future Proofing

Rather than future proofing in terms of creating features that will anticipate technology changes, we will be using the term future proofing to mean developing features with future features in mind. By writing our features with this concept in mind, all of our features were written in ways that allowed them to be easily changed, maintained and integrated.

No feature needed to be rewritten in order to integrate it with other features during the entire project, which is something that has often happened in other courses. This result is due to many group decisions, such as using Jira, establishing the issue Backlog and discussing implementation of features. We successfully avoided having to refactor endlessly which allowed us to develop additional features.

7 - Conclusion

We believe that the project was an overall success. This was due in large part to our frequent communication and efficient meetings, where issues would be raised and subsequently dealt with in short order. Furthermore, our flexibility on the approaches we could take for each issue led to those issues being evaluated by the majority of the team members at once, without butting heads. Responsibilities were doled out according to our skills and knowledge, such that everyone’s strengths were used to their fullest.