Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Software Design Specifications

System Architecture

High Level Block Diagram

The data flow through the system from the user's perspective can be seen in Figure 1 below.

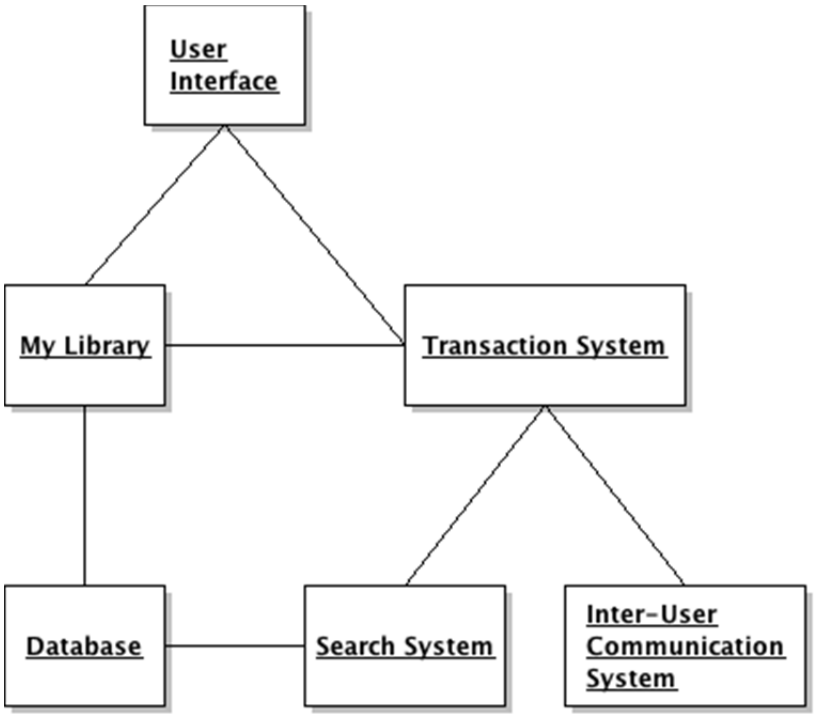


Figure 1: Data Flow (from user's perspective)

The user interface of the system connects to two main components: the "My Library" system, which allows a user to see the current state of his or her library, transactions, and loans; and the "Transaction" system, which allows the user to exchange books with others.

The "My Library" system allows for three functionalities: "My Books", "My Transactions", and "My Loans". "My Books" consists of all of the books the user owns, and users can add or remove books. "My Transactions" consists of all of the transactions in which the user is either an Owner or Client. And "My Loans" consists of all of the transactions that resulted in a loan, for which the book is currently not in possession of the Owner; that is, all the books a user is currently lending or borrowing.

The "Transaction" system is the means by which users exchange books. It interfaces with "My Library" to determine information about users' books, the "Search" system to allow the user to look up books in the

database, and the "Inter-User Communication" system to facilitate the transaction between two users. The "Inter-User Communication" system gets input from the Client and Owner to determine the conditions under which the owner trades, sells, or loans his or her books.

Diagrams

UML Class Diagram

The SharingBooks Class Diagram can be seen below in Figure 2.

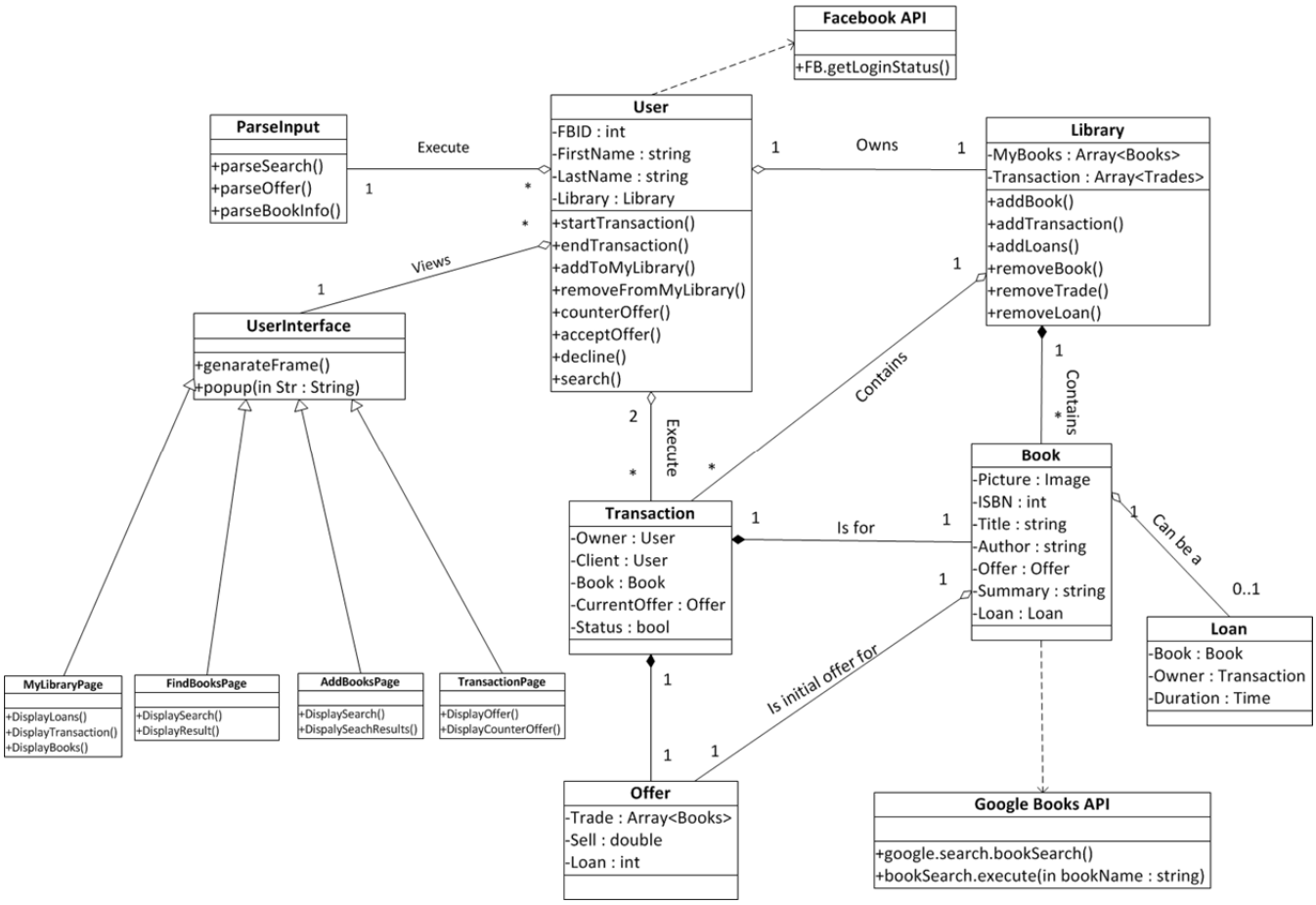


Figure 2: Class Diagram

The UML class diagram contains details about all of our major classes and their interactions within our system.

- At the core of our class design is the **User** class. The User class stores the customer's personal information such as their Facebook ID and name, and also contains a Library object representing

that customer's personal library. To retrieve the customer's information, the User class talks to the Facebook API and opens a session when it gets a response from Facebook that the user has logged on. The User class also has methods that are called when a customer wants to perform a use case action, such as starting and ending a transaction, countering and accepting offers, searching for books, or modifying their personal library.

- The **UserInterface** class is the base class for the following four classes: **AddBookPage**, **FindBooksPage**, **AddBooksPage** and**TransactionPage**. This class decides the static part of the display. generateFrame methods displays the static item such as logos and sidebars. Popup method handles static part of popup pages.

- The **AddBookPage**, **FindBooksPage**, **AddBooksPage** and **TransactionPage** class extend UserInterface class and they handle creating display for "Add Books" page, "Find Books" page, and "Transaction" page respectively.

- The **ParseInput** class takes all the information added by the user and send to business logic side so we can add to database or vice versa.

- The **Library** class contains arrays for Book objects, Loan objects, and Transaction objects. These represent the current books, loans, and transactions that the customer has. This class also has methods to add and remove objects from its arrays.

- The **Transaction** class represents a single transaction between two Users. This class contains fields representing the owner of the book involved in the transaction, a Book field representing the book itself, a Client field, a CurrentOffer? field with an Offer object, and a Status field representing the current status of the transaction and whether it is currently ongoing or has ended. The Offer object in the Transaction class contains the current offer details of that transaction.

- The **Book** class represents a single book. It contains data about that book, such as its cover image, ISBN number, title, author, and summary. If this data is not currently contained in our database, the application will contact the Google Books API and perform a search to look up information on that book. Finally, the Book class also contains an Offer object. This Offer object represents the initial offer details for this particular book.

- Since books can be loaned in our system, there is also a **Loan** subclass that contains all the fields from the Book objects, as well as the owner information from the Transaction class and the duration length of the loan.

UML Sequence Diagrams

Use Case Scenario 1: Adding a Book

User begins the SharingBooks app on Facebook and wants to add a book to his/her library.

Team D.A.W.G. Squad

Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Diagram

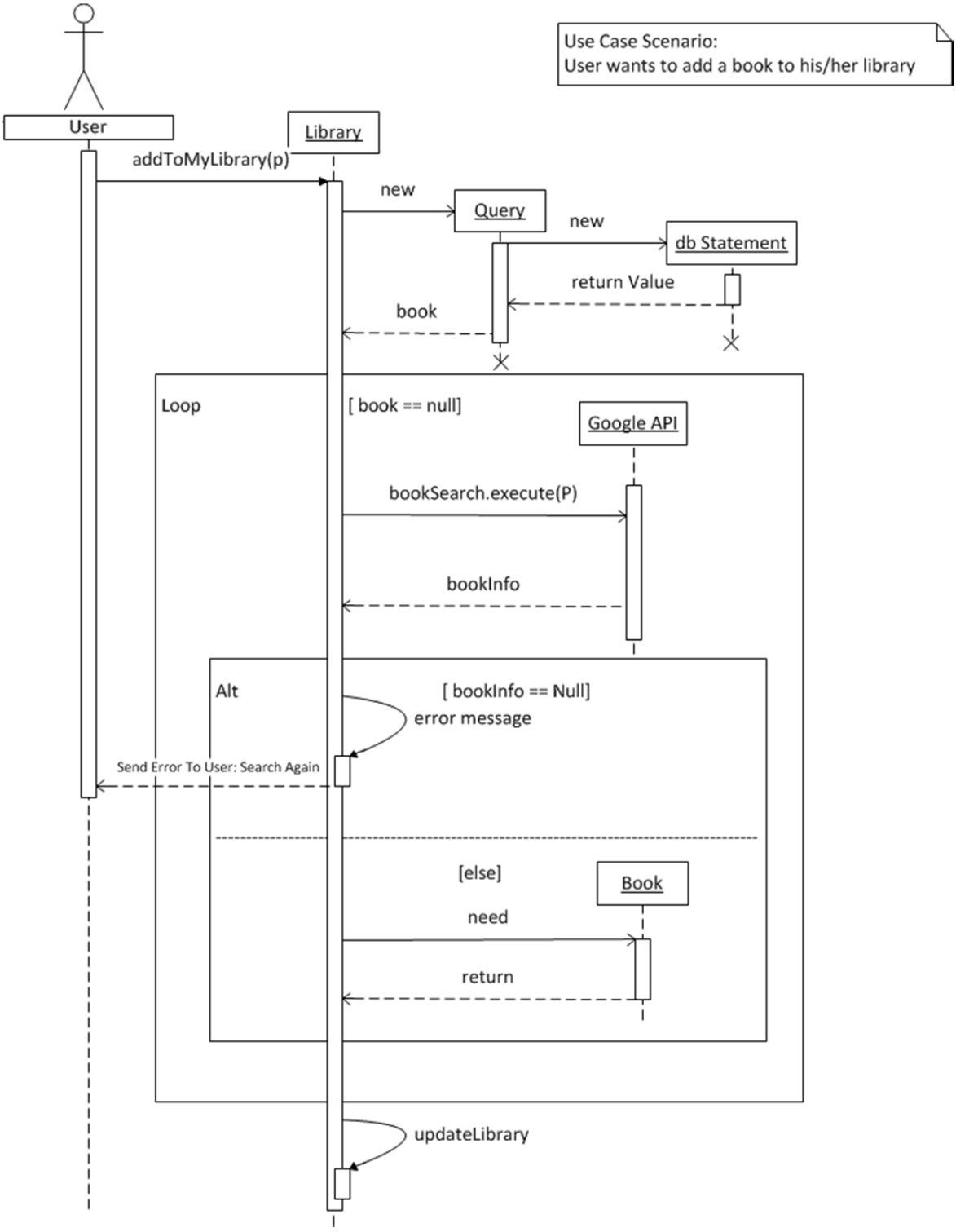


Figure 3: Scenario 1 - Adding a book

Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Pseudocode

```
/* Checks if book is in local database */
Create new Query for User specified book
Query local database for that book
Initialize Book with query results

/* Queries Google Books until book is found or User quits */
WHILE Book is undefined

      Query Google Books database for that book
      BookInfo = Query results

      /* Query results will have to have enough information
         to populate BookInfo. If it's not enough, BookInfo
         will be undefined */

      IF BookInfo is undefined THEN

         Output an error message to user

      ELSE

         Initialize Book with query results

      ENDIF

ENDWHILE
```

Explanation

To realize the scenario above, a user starts with calling the function: addToMyLibrary(p). The parameter p indicates the book that the user wants to add to his/her library.

After the function gets called, it creates a new query for the user specified book to check if the book is in local database.

If the book does not exists in the local database, the function then queries Google Books using Google Books API until the book is found.

If it still returns Null, the system assumes no such a book exists. Thus, send error message to the user.

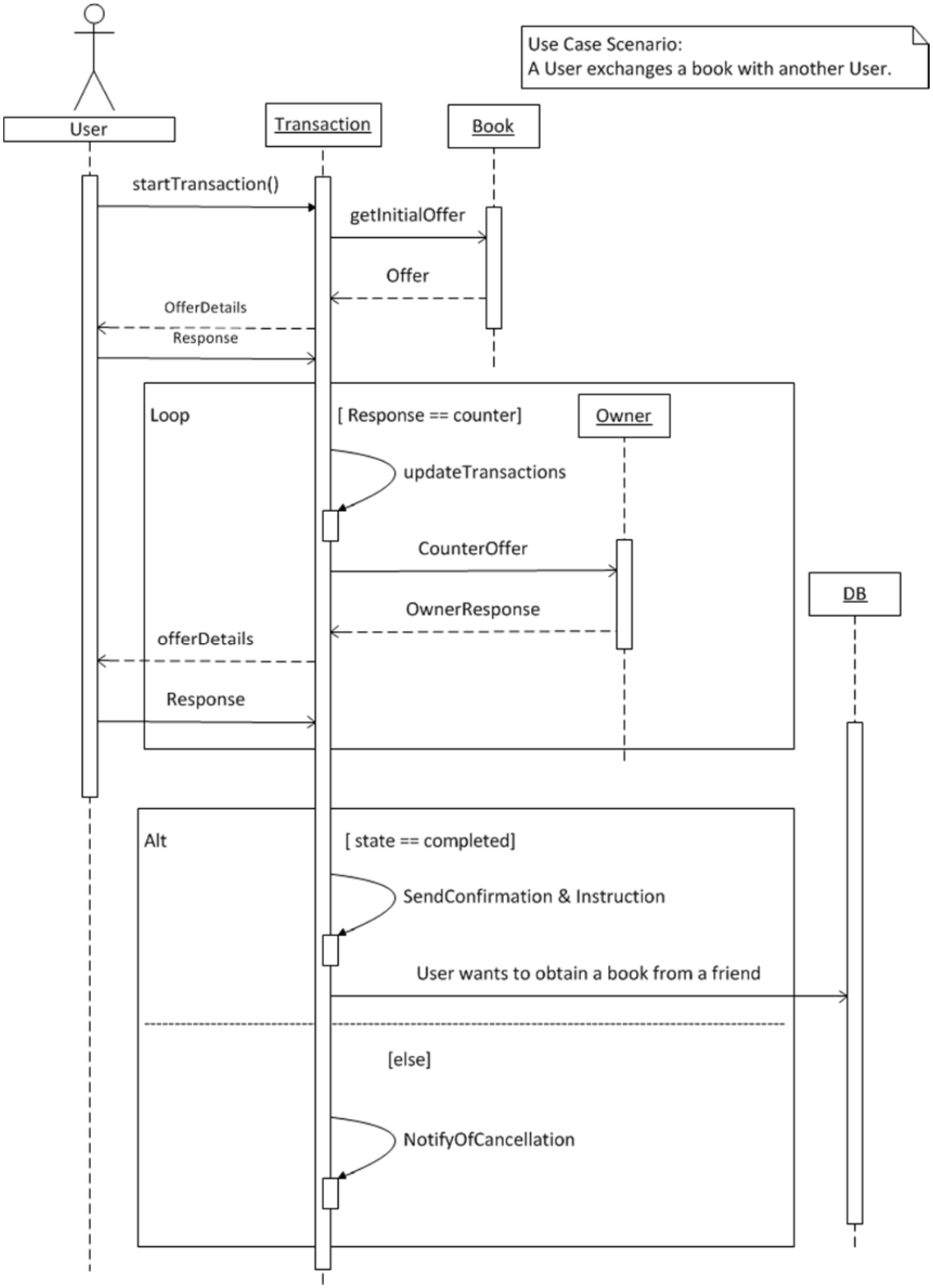
If the book is found, it adds the book by initializing the book with query results.

Use Case Scenario 2: Exchange a Book

A User exchanges a book with another User.

Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Diagram



Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Figure 4: Scenario 2 - Adding a book

Pseudocode

```
/* Initiate transaction */
Create new Transaction between Client and Owner
Get initial offer from Owner for Book
Notify Client of initial offer
Get initial response from Client
Set initial status of Transaction
/* status will be canceled, pending, or completed */

/* Go back and forth between parties until Client
   agrees on Owner's offer */
WHILE status is pending

      Update Transaction with Client response
      Notify Owner of Counter Offer
      Get new response from Owner
      Notify Client of new response
      Get Client new response
      Update status of trade based on new Client response
      /* status will be canceled, pending, or completed */

ENDWHILE

/* At this point, the transaction is no longer pending.
   It is either completed or canceled. */

IF Transaction state is completed THEN

   Send offer confirmation to Client and Owner
   Instruct Client and Owner on how to exchange Book

ELSE

   Send cancellation confirmation to Client and Owner

ENDIF
```

Explanation

To realize this scenario, a user start with calling the function: startTransaction(). This begins the transaction between a client and an owner.

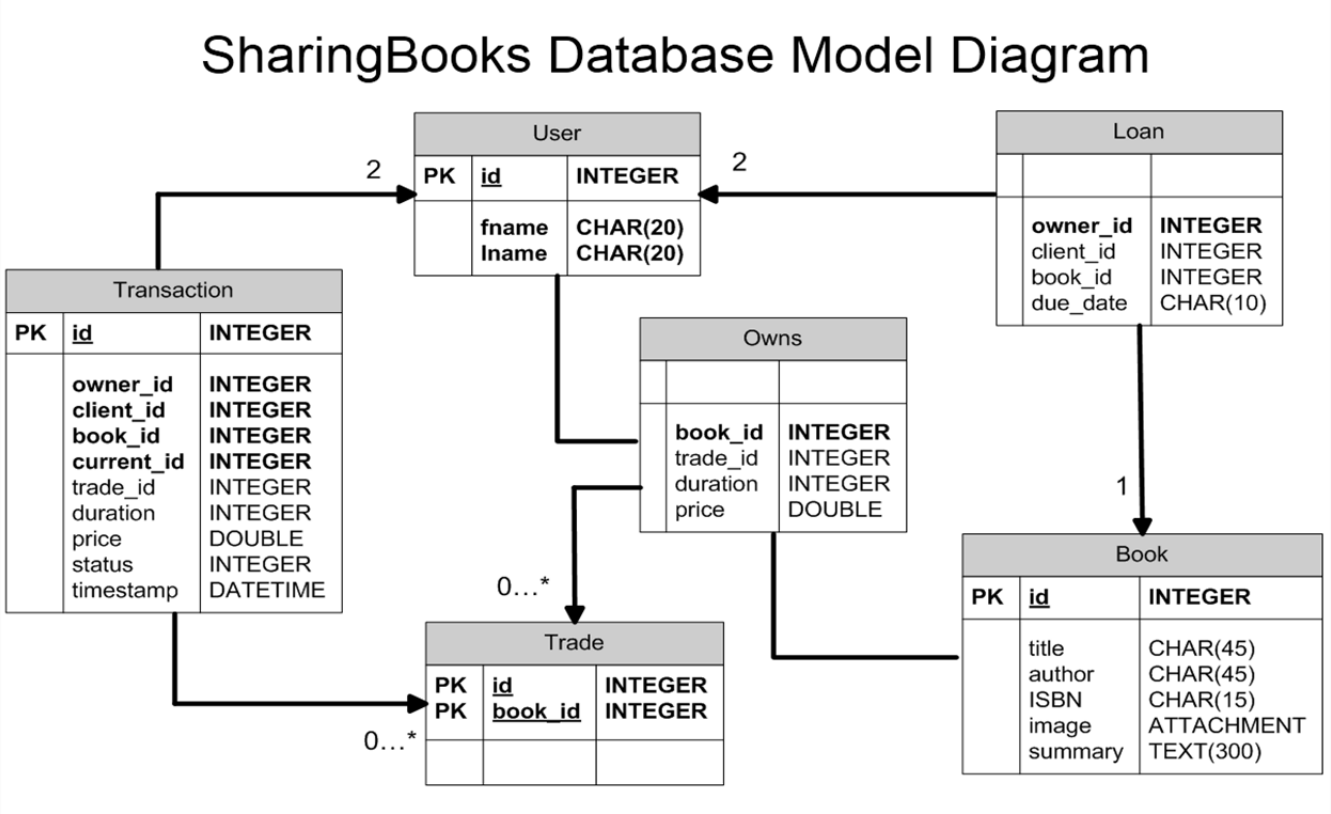
Transaction class then gets initial offer from the owner for the book and notifies the client of initial offer details. Then it receives the initial response from the client.

If the client user does not agree with the offer, he/she responses with a counter offer to continue transaction. In this case, the transaction updates itself with the client response and notifies the owner of Counter Offer to get new response from the owner.

When the transaction receives the response from the owner, it sends the offer details to the client and he/she responses to it. This transaction continues until the client confirms the offer or either one of them cancels.

Finally, if the client agrees with the offer, the transaction class sends offer confirmation to the client and the owner and instructs the client and the owner on how to exchange the book. Or, if the client or the owner cancels the transaction, the transaction sends cancellation confirmation to Client and Owner.

Database Model Diagram



The database model diagram shows the design of the MySQL database used by SharingBooks to keep track of users, the books they own, and the current transactions, possible trades, and outstanding loans of books between users.

Highlights of the design:

- Each user is represented by a tuple in the **User** relation, which includes a user id that is unique system wide and is used to track which users are involved in which transactions.

- **Book** contains a tuple for each distict book that has been added to a library by a user, with data populated from the Google Books API. Each book has a unique book identifier, and all instances of a given book elsewhere in the db refer here for book details.

- **Owns** keeps track of the libraries of each user, with an entry for each book. When books are inititially added to the library, a tuple is added here and the trade_id, duration, and price fields are populated with the initial values provided by the user (i.e. what the owner would initially accept in exchange for the book).

- A tuple in the **Transaction** relation is created every time a user proposes a new offer for a book, and tracks the two users involved in the bartering, the current offer being made (as well as who made it), and the book. When an offer is accepted, the status goes to complete, if it is rejected then the status goes to rejected, and otherwise the status is pending.

- The **Trade** relation acts as an auxilary list of the books that are listed as acceptable in trades that are being offered throughout the system.

- The **Loan** relation receives a new tuple whenever a transaction completes that involves a 'duration', and then maintains a record of the owner of the book, the borrower, the book, and when the book should be returned.

Process

Team Structure

Since our SRS we have re-arranged the team structure to better accommodate our system architecture. We initially thought that our project would simply be split into a front-end Facebook application, and a back-end database, with logic to integrate the two. This integration logic was very loosely defined. Now that we have thoroughly considered exactly how each aspect of our system is going to interact, and what the logical flow of information will be through our architecture, we better understand how to divide up the work to achieve our design goals.

The notion of front-end and back-end has been expanded into a model, view, controller (MVC) architecture. The model will be our database which stores all the data for our application. Setting up the model will require writing MySQL scripts to set up the tables. This "Model Team" is now composed of Jedidiah and James. This will be completed early in the process to facilitate testing the controller/model interface. The controller is the business layer that controls the flow of data between the UI and the database. It will contain the significant portion of our application’s logic. This "Controller Team" is now composed of Wei-Ting, Troy, John and Tatsuro. The view can generally be described as the UI. It can be split into two main parts, one of which displays the content for the user to view, the other part which handles user input. This "View Team" is now composed of Ken and Greg.

Team D.A.W.G. Squad

Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

For the Zero Feature Release, James and Jedidiah will set up the database tables and populate it with dummy data. Wei-Ting will register our application on Facebook. John, Troy and Tatsuro will start learning CakePHP and set up basic class outlines. Greg will develop a basic web page to be displayed and host it on Ken's personal server. Ken will write the script that will build our system and allow the user to access our web page from a Facebook account.

There will need to be a lot of coordination between groups to put this architecture together successfully. Wei-Ting is our project manager, and will ensure that each group is kept on track. He will also handle most of the client requests and feedback, which he can translate into action items for the group. We have set up a wiki, which is where all documents are being kept. Also, we are using Mercurial for version control. Weekly client meetings will take place on Thursdays at 7:00. Status reports will be submitted by midnight every Sunday.

Project Schedule

A sample timeline can be seen in Figure 5 below:

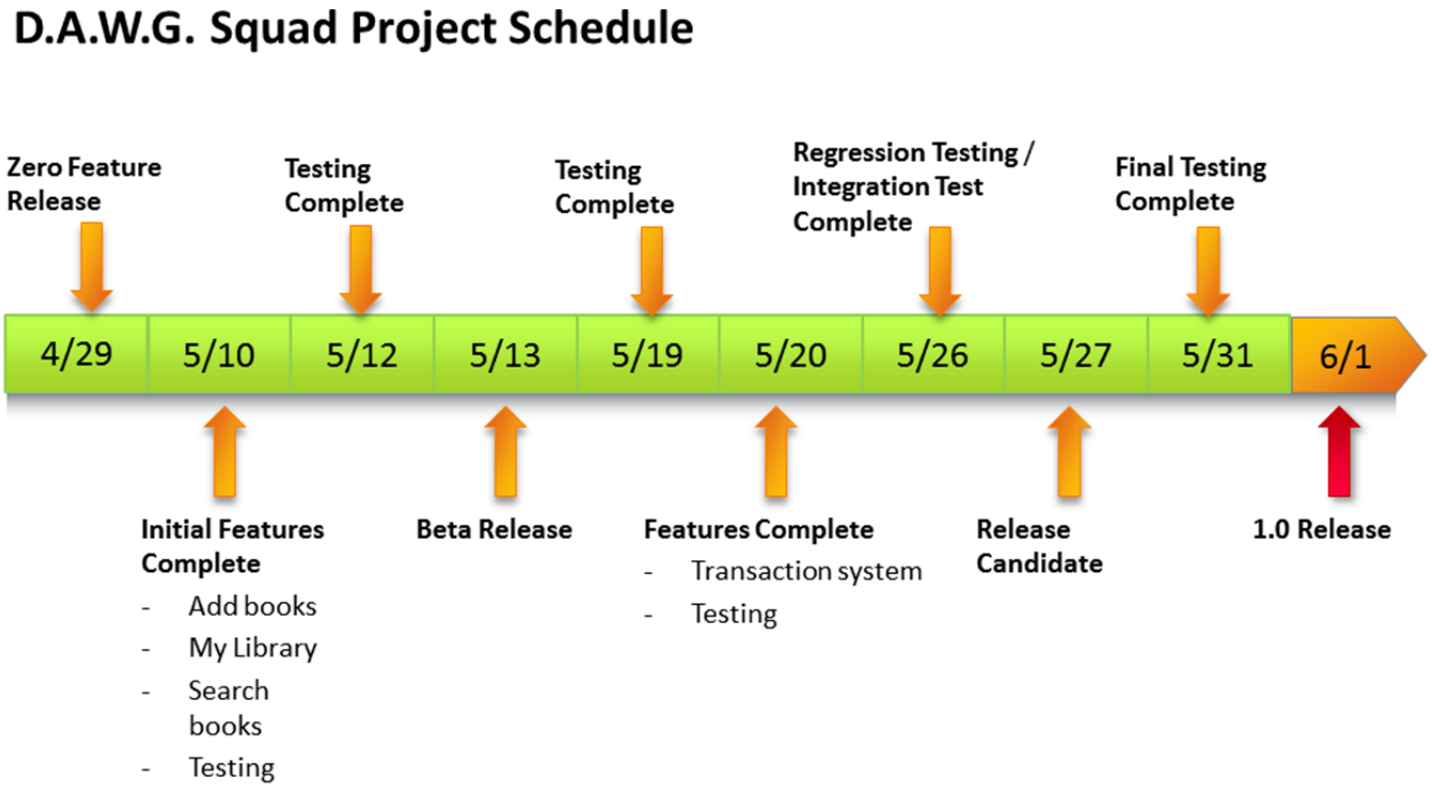


Figure 5: Project Schedule

Below are detailed descriptions about each date in the timeline above. Note that each milestone depends on the previous milestones being completed (except testing). Testing for features can be done in

concurrent with the feature being implemented. Single date means to be finished on that date while a range means the estimated development time.

Zero Feature Release - April 29th

- Implement installable app from Facebook (Wei-Ting, John, Tatsuro, Troy)
- Implement database tables (Jedidiah, James)
- Install script for default Linux VM (Ken, Greg)

Begin skeleton of My Library feature - (April 30th - May 1)

- Implement basic UI and functionality of My Library
    - UI (Ken, Greg)
    - Business logic manipulating My Library (Wei-Ting, John, Tatsuro, Troy)
    - Database for My Library (Jedidiah, James)
- Further feature releases will build on this feature as we go

Initial Testing for My Library feature - May 1

- Perform extensive testing on My Library feature (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)
    - Black box test database (Tatsuro, Troy)
- Document bugs and fix priority bugs (All)

Implement Add Books Feature - (May 2 - May 3)

- Implement UI for Adding Books (Ken, Greg)
- Implement Business Logic for Adding Books (Wei-Ting, John, Tatsuro, Troy)
- Implement database for Adding Books (Jedidiah, James)

Add Books Feature Testing - May 4

- Perform extensive testing on Add Books feature (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)
    - Black box test database (Tatsuro, Troy)

- Document and fix priority bugs (All)

Implement Search Books Feature - (May 5 - May 9)

- Implement UI for Search Books (Ken, Greg)\
- Implement Business Logic for Search Books (Wei-Ting, John, Tatsuro, Troy)
- Implement database for Search Books (Jedidiah, James)

Search Books Feature Testing - (May 10 - May 11)

- Perform extensive testing on Search Books feature (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)
    - Black box test database (Tatsuro, Troy)
- Document bugs and fix priority bugs (All)

Integration Testing - May 12th

- Start testing interface between elements of MVC framework (All)

Beta Release - May 13th

- Release for web (Wei-Ting)
- Install Script Done (Ken, Greg)
- Keep logic, UI, database updated (All)
- Document bugs (All)

Implement Transaction System Feature – (May 14th - May 18th)

- Implement UI for Transaction Feature (Ken, Greg)
- Implement Business Logic for Transaction (Wei-Ting, John, Tatsuro, Troy)
- Implement database for Transaction (Jedidiah, James)

Transaction Feature Testing - May 19th

- Perform extensive testing on Transaction feature (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)

    - Black box test database (Tatsuro, Troy)
- Document bugs and fix priority bugs (All)

Features Complete Release - May 20th

- Release for web (Wei-Ting)
- Install Script Done (Ken, Greg)
- Document bugs (All)

Regression Testing/System Integration Test - (May 21st - May 26th)

- Extensive regression testing for all features (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)
    - Black box test database (Tatsuro, Troy)
- Document bugs and fix priority bugs (All)

Release Candidate - May 27th

- Final implementation refactoring and bug fixes (All)
- Perform code reviews to identify any weaknesses (All)
- Install Script Done (Ken, Greg)
- Should be in a release-able state at this point

Final Testing Phase before 1.0 Release - (May 28th - May 31st)

- Extensive regression testing for all features (All)
    - White box test UI (Ken, Greg)
    - White box test business logic (Wei-Ting, John)
    - White box test database (Jedidiah, James)
    - Black box test UI (Wei-Ting, John, Tatsuro, Troy)
    - Black box test business logic (Ken, Greg, Jedidiah, James)
    - Black box test database (Tatsuro, Troy)
- Document bugs and fix priority bugs (All)

1.0 Release - June 1

- Make a few final additions and improvements to our Release Candidate (All)
- Release final version of our application on web (Wei-Ting)
- Install Script Done (Ken, Greg)

Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Test Plan

Unit Testing System:

Based on the System Architecture this project will have three major layers of testing:

- Testing the Model (Database).
- Testing the View (UI / Content Generation).
- Testing the Controller (Business Application Logic)

Each layer plays a significant role in the functioning of the software and tests will be performed on each component of the MVC framework before their integration is tested(see System Testing below).

For performing Unit Testing for each layer, the team will follow two testing strategies: white-box-testing and black-box-testing against every method or function that is being implemented. The team structure as described above involves three core-teams: Model, View, and Controller. All of these teams will create White-box tests as they implement every feature by classes or Database tables in their end of the code. The developers in each team will have understood key aspects of their implementations and would be writing tests to ensure good coverage. Simultaneously, the View Team and Model Team will write Black-box tests for the Controller Team without having any idea of their implementation. The Controller Team will create Black-box test for both the View Team and Model Team. Black-box tests will be based only on the SRS and will satisfy all preconditions that are set. These tests will be significant in testing boundary and corner cases. Each of these tests will be developed concurrently during the implementation and coding phase. All these tests will be scripted to automate the testing process, developers will be given flexibility to use any scripting language of their choice, and will document the process so other team members can run any of the tests also. The White-box tests will be run at as each feature in the project is being developed and the results will be documented. The Black-box tests on the other hand will be run with lesser frequency than the white-box tests and will be used initially during the development to test boundary cases and after the development of feature to test if that feature was successful.

System Test (Integration) Strategy

The team will write Integration tests to ensure that the communication between the Model, View, and Controller is as hassle free as possible and that they really work together. For this at least one team members from each team will create tests to perform this very task. These tests will be scripted and created to be automated to run every day during testing phase (usually before each releases, please see project schedule for more details) with test results stored regularly every night. The purpose of these tests is to ensure that the interfaces between each part of the model-view-controller framework are functioning correctly. Two main issues that we have identified as of right now are:

1. Consistency - between the database, business logic, and UI. For example: run a DB query (acting as the front end) and see if the results from the database would be as expected and stored in the correct DB tables corresponding to My Library.
2. Handle multi-threaded issues: to ensure that the system can handle user logged into multiple browsers and also situations when multiple users are trying to use the application. For example: create tests to specifically simulate multiple user accounts and ensure that each library is reflected consistently.

Usability Test Strategy

Usability will test the View Team's code for layout, usability and expected behavior. The Build will be tested at least once a day by a different member in a rotating schedule. This will ensure that issues will be found quickly and that we get different opinions on layout and usability as these are dependent of the user of the program. Tester will look through the use cases that have been implemented and complete task on the current build. They will note any deviations from expected behavior, and any opinions on usability and layout. Deviations from expected behavior will be fixed immediately. If more than one tester mentions the same usability issue or layout they will be reviewed and potentially redesigned.

Adequacy of Test Strategy

As laid out above we will be testing the program in a number of different ways by more than one person, multiple times a week. This will provide multiple chances to catch bugs so that we can get them fixed quickly. As long as we follow the plan as it laid out the testing strategy should be adequate.

Bug Tracking Mechanism

Bugs will be tracked using Google code page that has been set up for project. Once a bug has been reported on the Google code the team will work together to track the bug. Once the bug's origin is determined then the PM will appoint someone to look into the issue and fix it or hand it off the proper programmer if they are unable to.

Risk Assessment

Facebook Changes the API

There is a medium likelihood for this occurring. This could potentially have a major impact on our project. If we are nearing the end of the project, any changes that break our application could be disastrous. Hopefully if any changes occur, they occur earlier in the development process so we have time to modify our application accordingly.

The issue of Facebook changing their API was brought to our attention by word of mouth. After turning in our SRS we began looking online and found various blogs/posts where developers complain about their

code breaking after an unannounced API change. However, these posts happen very soon after the change, so this might be a good place to start looking for solutions.

The View Team and Controller Team developers will be testing their code often to detect any issues early. This will allow us to address fixing any bugs as soon as possible. This will also help control the affect such a change has on our application and reduce the down time. Developers can also check the Facebook developers blog to stay abreast of all notifications and changes.

To mitigate this risk should it occur, the developers can search online for fixes to bugs caused by Facebook changing their API. The View and Controller Teams should also consider halting and further development until the bug is addressed and corrected.

Integration between Model, View, and Controller fails

There is a low likelihood of this occurring; however, the impact to the project will be very high. The risk of this occurring was really brought to our attention by Professor Ernst in one of his lectures. He really emphasized the potential hazards associated with this process and encouraged us to give due diligence to the design phase of the software development lifecycle.

D.A.W.G. Squad put a lot of thought into how to design our architecture to facilitate easy communication between the database and the user interface. We settled on a model, view, controller architecture. The model will be our database that stores our data, the controller will be the business logic that coordinates communication and contains the majority of the applications logic, and the view will be a combination of the UI and user input processing. While we were designing our architecture, group members were in constant communication with each other. In essence, we designed all three elements in parallel which allowed us to address the needs of the three architectures. After the database tables are set up, we will inject test data that the view and controller teams can test accessing as they develop the application.

We do expect there will be some communication problems between the three elements of our system, but we plan to mitigate this risk by establishing an interface early on, and address any problems that arise early and quickly.

Users pursuing legal action due to lost property

The likelihood of this occurring is low. The impact on our project if this does happen is medium to high. The issue here is that we are providing a means for students to loan their textbooks to their peers on Facebook. If a user loans their book to a friend, and the friend does not return the users property, they might seek legal action against our group since we provided the service.

It is not at all uncommon in today's society for someone to sue some company, internet or other, for something bad happening as a result of that person using the companies product or service. This is why it will be very important for D.A.W.G. Squad to protect themselves. A plausible solution would be to include

a disclaimer that basically says “use at your own risk”. Users would have to agree to this before they could use our application.

Even with disclaimers, legal action happens. In the event this occurs, we would have to seek legal representation. If we made sure to be thorough in our disclaimer wordage, we will get through it just fine.

Our Application is just too similar to already existing alternatives to become popular

There is a medium likelihood of this occurring as well as a medium impact to our group. Before our SRS was due we started researching already existing applications that were similar to ours. We discovered a few, and have since discovered a few more. We spent a long time discussing how we could make our application unique , intuitive, and yet be able to complete our application by the project deadline.

Our solution to ensuring our application is genuine is to include the ability to barter. We feel this will be something similar book sharing applications do not have. Also, we have put a great deal of effort into designing our UI to be as intuitive and user friendly as possible. We really engaged our client during the paper prototype user interaction test, and made changes based on their feedback.

There is always the potential that even given the unique features we plan on implementing, users still don’t use our application. This might be a result of our application not being unique enough, not intuitive enough, or just plainly not useful enough. Additional usability testing can help mitigate this risk.

Application performance significantly degraded due to CakePHP framework speed/scalability issues

There is a medium likelihood of this occurring and this will have a medium impact on our group. After doing a lot of research on different frameworks to use to develop our Facebook application, we found that CakePHP seemed the most straightforward, with the most documentation and examples. The problem is that there are also many posts by developers complaining about it being too slow. Although this slowdown will probably not affect our application initially, there might be scalability issues. As the number of users grow, the sluggish framework might not be able to handle the data request load by end users in an efficient way. This will make our application very unattractive to users.

By keeping the operations of our application pretty simple, and not making numerous large data requests to the back-end database, we should be able to maintain acceptable performance. To test this, we will have to simulate numerous user inputs. These tests will allow us to get a sense of how well our application will scale, and will be especially useful after each feature is implemented. This will allow us to track the performance of our application as it grows.

To mitigate the risk should it happen, we will look for ways to improve the efficiency of our application. This might include things like using a code analyzer, making sure the common cases are fast, and code refactoring to relieve bottle necks.

Team D.A.W.G. Squad
Greg Brandt, Ken Inoue, Jedidiah Jonathan, Wei-Ting Lu, Troy Martin, Tatsuro Oya, James Parsons, John Wang

Documentation Plan

Currently, our application will include a Frequently Asked Questions (FAQs) page as well as a dedicated help menu for our users to access when they are using our application. In addition, there will also be a simple tutorial for using our application, which includes introduction to adding, searching, and making transaction with books.

Coding Style Guidelines

We plan to use CakePHP, and their coding conventions can be found here: http://book.cakephp.org/view/901/CakePHP-Conventions

In addition, we will use JQuery Javascript library to do animation and Ajax interaction with CakePHP. Their coding convention is found here:http://docs.jquery.com/JQuery_Core_Style_Guidelines

These are the coding style guideline that everyone on our project will follow. We plan to enforce these guidelines by having weekly code reviewing sessions with everyone. This serves as both a way to enforce these guidelines as well as a way for everyone to understand our project on code level.