Cs611 bank OOD documentation

@Tingzhou Yuan @Shu Xing
@Anoohya @Srinivas

Introduction @Tingzhou

The project was supposed to be a 4-people collaboration project, so the key to effectively separate tasks between team members was to divide the project on multiple aspects (AOP). In our case, we're making a single app that runs on one single machine, where the user interface fetches data from the backend through direct Java method calls. This means there's no need for an extra authentication layer like we do in HTTP where requests are first validated before they are allowed to hit the backend endpoints. We exercise the MVC pattern which emphasizes separation of model, view and controller.
Data persistence is, as requested, done through an integrated SQLite3 database. We use an ORM (object relational mapping) package (ORMLite) to deal with the data accessing objects, which significantly simplifies design as well as implementation. Introducing relational concepts in our project means that the object designs should be different from one that completely runs in host memory because everything now revolves around the persisted DAO instead of actual objects.
The project specifications asked for interest to be generated since the bank takes savings. We decide that in order to properly demonstrate this function as well as timed deposits/loans, we would need a simulated time that could be either accelerated or decelerated on demand. Since we are no longer pegging on real world time, this introduces some challenges and deeper into the document we'll discuss some of our solutions.
Java has long had the problem of checked/unchecked exceptions, in this project we attempt to address these problems by introducing the Result<T> class, which indicates the specific state of an operation call on the service interfaces. At the end of this documentation we discuss some of our conclusions on this topic.

Implementing the the MVC Pattern@Tingzhou

MVC has low coupling. It separates the view layer from the business layer, which allows changes to the view layer code without recompiling the model and controller code. In our code, we use MVC to help us separate BankUI, the actual frontend implemented with Java Swing, and bankBackend, which provides abstract, stateless services APIs to the frontend..

In practice the controller layer may be further divided into service and controller, where service takes care of business logic while controller deals with possibly unsanitized user input. In our project, since everything is done through Java method calls, the actual controller layer is not necessary and our service package is in fact the controller. Therefore our backend includes a

data model, service layer and controller layer.If we change the business process or view, there is little correlation between them.

We'll discuss the details of our object designs in the frontend and backend in the following paragraphs.

Frontend (View) @Shu

We use netbeans to write and manage bankUI. Through netbeans, we were able to make UI design with a visualized editor. If we want to make some changes to GUI, we can use interface management and parameter binding, this has no impact on the backend.
We use object oriented design to implement frontend, the pages are extended from JFrame, JPanel and other basic components of JSwing. We use JFrame to implement the main interface, the panel in the frame is based on JPanel. Meanwhile, JLabel, JTable, JButton and JList are used to fill in the panel.

Components come with data: @Shu

Our UI pages include multiple components. When the interface needs to show specific data, the component will actively call data through specific parameters according to page requirements. Instead of passively accepting data from the backend in some way. Components with data added to the pages can make users track data in time. When a user reloads the page or makes some action through the interface, the corresponding components will update the latest data by actively calling some parameters and functions.

Managing context while dealing with netbeans @Tingzhou

The problem with netbeans is that for each of the components designed with the UI designer, it couldn't have any arguments in its constructor, effectively making it somewhat stateless. However, in order to properly reuse these components, we had to come up with a way to pass context like "Current user" and "Current Account" to the components. To tackle this problem, we made a centralized UIContextManager class which allows users or accounts to be bound to different JFrame or JPanels upon initialization and deregister themselves upon the dispose call. This work around proved very effective in developing cascading UI components.

A common component for user dialogues @Tingzhou

A common scenario in UI development includes a call to a backend endpoint, if the call succeeds, the UI may get updated, if the call fails, there has to be some way to pass information back to the user. In our project, we **encapsulate** the JPotionPane's showMessageDialog function into a single component: AlertUI, which allows developers to send user success or error messages. This allows for greater reusability and makes development easy.

Reusable components: @Shu

In our design, some parts of the code can be reused in several pages to support operation of the interface. Allowing a component to be reused multiple times reduces the workload of page development and implemented object oriented design.
We only need one component to meet the needs of many interfaces. For example, we have combobox, which is a component of currency selection. Meanwhile, it is also used in other classes to fulfill the page as there are other places that need to use the combobox to make some choices. Apart from this, account selection can also be reused because in many pages the user can select the account they want to enter. The project also has a management panel, to satisfy the management function of the manager. This is reused whenever a manager wants to do some operation to the bank, at that moment the interface needs the management panel.

The classes we used in NetBeans for frontend part are @anoohya

Home UI
This displays the balances in all the accounts that the user has on savings and checking accounts. Also, You have the access to the deposit, withdraw, modify account status and transfer money pages.

Authentication UI
In this page users will be able to type the email and login password or create a new account. So that they can access the homepage of the bank.

Stock UI
This displays the present stocks and prices of it. It shows the buy and sell options.

Account summary
All the details of the accounts are shown here.

Account state UI
Here you have the option to change your account status to open, close and inactive.

Transaction builder UI
Transferring the amount from one account to another happens through this page. You can select the account that the user wants to send to and from which account, give input the amount of money. Users can also give a message to remember the purpose of the transaction.

Loan account Ui
It displays all the current loan plans and the rate of interest. To borrow money we have to get to this page.

Saving account UI

The deposit will be paid through this page. So, it displays all the previous deposits that the user has made and also the current interest rate.

MoneyIOUI
If we are withdrawing the money, the user has to input the amount of cash that they are willing to take and give the mode of currency.

Integration between UI and backend @srinivas

The same classes are used in two different files for integration. The above classes will use it as UI class components. The data is initially saved in SQL when a user opens an account. We are pulling information from SQL for the purpose of logging in. We'll open a checking and savings account so we can make deposits and withdrawals. If the person has more than the security account limit, then the user can open a security account. Without a security account, a user cannot purchase or sell stocks for trading. Based on past history, the manager will set the interest rate and provide the loan amount to the user. The manager of our bank project can download reports as needed.

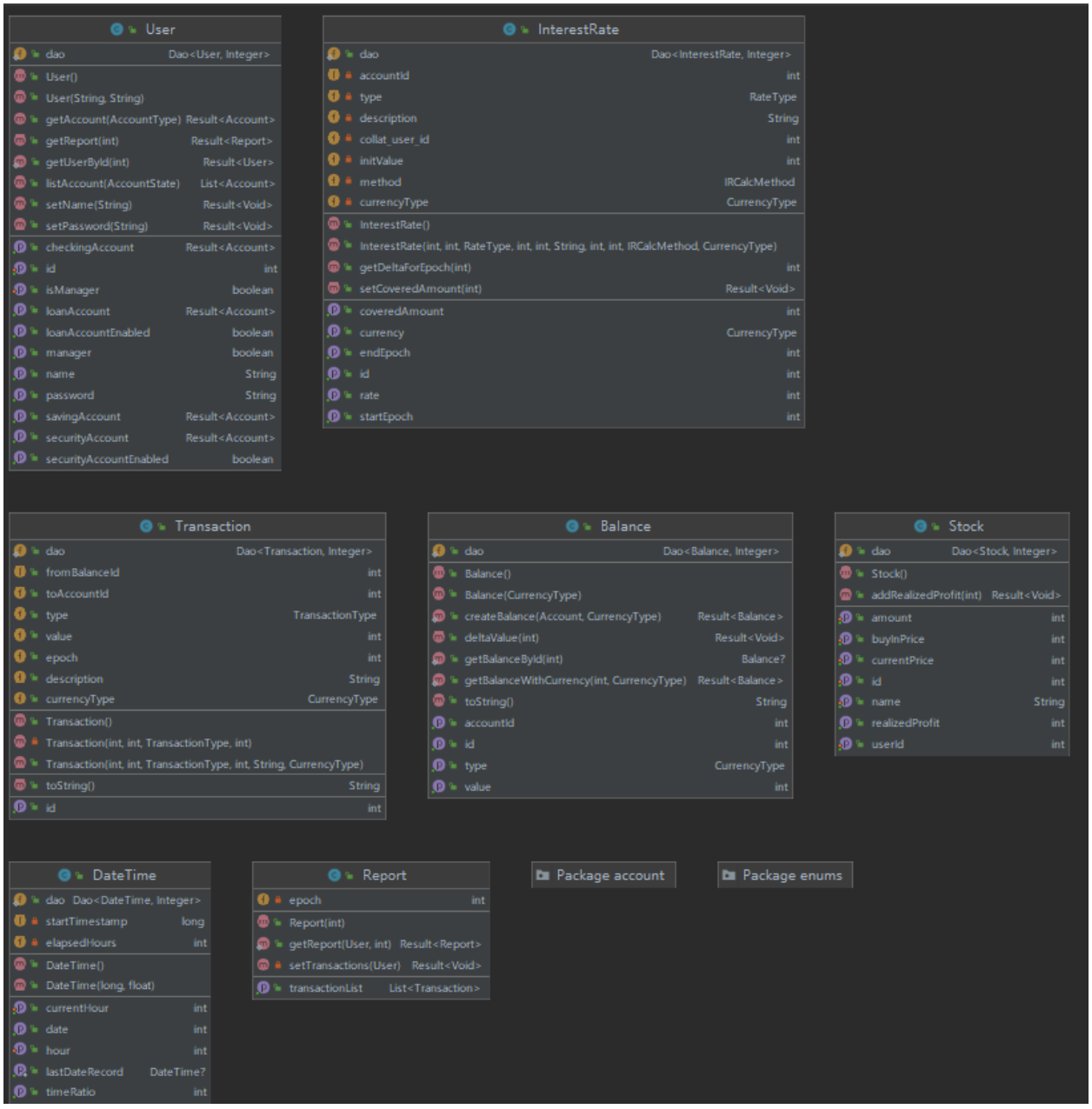
Backend (Data model, service and controller) @Tingzhou

In the Backend, we have four main packages: dao, entity, factory and service.
The backend uses the dao layer to implement access to persistent data. Through access to the data interface, we can implement data’s storage, modification and query.
The service layer is mainly responsible for the application logic design of the business model. In this way, we can call the service interface in the application for business processing.
We’ll discuss each of them in order.

The dao package

DaoManager In projects without external ORMs, this package serves as the layer where business logic codes call objects in this package to fetch data from a database. In our case, since data access objects are provided by the ORMLite package, we only had a DaoManager **singleton** here that deals with the creation of tables, and actually managing the dao objects.

The entity package



The entity package contains all the object types that should be persisted in a database.
User Due to time constraints, our user object is extremely simple, with little attributes that characterizes a user. However, this does not impact system functionality and we decide that other attributes are simply unneeded in this context.

Managers are a special type of users that has access to the manager dashboard, which allows them to control system settings, interest intervals and even other user's accounts. Users distinguish from Managers through a special boolean attribute "isManager". Any user can become manager if they please.

Account In our case, since all the user's accounts share the same characteristics such as being able to hold arbitrary amount of money of any support currency, we put all the accounts under one account parent class, each distinguishable by just an AccountType enum, whereas the inherited ones (**Checking, Saving, Security, Loan**) only have domain-specific behaviors that implement business logic, such as generating loan interests or saving interests. We find that this significantly simplifies design and makes the codebase flexible. Accounts relate to their owner (users) by having the foreign key user_id in the object relation.
Accounts also have the state field with the AccountState enum indicating whether the account is closed, open or frozen. In this way we can easily charge money on account openings/closings since there's only one endpoint for it. In a sense, this entity implements the **State** pattern.

Balance The same argument holds true for balance as well, since the onlything that separates one balance from another is the type of the money it stores, we make all balances distinguishable by the CurrencyType enum. Balance objects relate to the account actually containing itself through foreign keys in the object relation.

InterestRate When designing the objects for implementing loans and savings, we find that these two share identical business logic, with the only difference being that one has interest paid by the bank, while the other one has interest paid by the users. Thus, we create an extra layer of abstraction and make Loan and Saving logics share the same InterestRate object, distinguishing themselves through an enum RateType. We find that later implementation significantly benefits from this design as each row of interest rate stands for one saving/loan request, and as they share identical features, little new code had to be written. Additionally, we considered how interest rates can be calculated in many different ways, and in this project we provide two: simple and compound, where loan orders are calculated as compound interest while saving orders are calculated as simple interest, to demonstrate the difference.

Transaction This is the most important entity in the entire bank app in our project. Any type of money transfer, interest generation, money deposit/withdrawal, stock purchase… all needs to be abstracted in to a transaction. In this way, we can maintain the consistency of the system by arguing that as long as all the money goes through a Transaction, the total amount of money in the system is an invariant, as long as it's implemented correctly. One may consider transactions in our system a type of **Command** pattern, since every money movement is encapsulated by the transaction object, and that we need only one Transaction handler endpoint to do the job.
The transaction had six critical attributes:

- fromBalanceId from this field we can deduct the actual type of currency that's being transferred since one balance is only associated with one currency type. We can also make pre-flight checks on whether the sending side's account is actually open and has enough money to complete the transaction.
- toAccountId from this field we can check the receiving end of the transaction. Note that we didn't use the balanceId because in our design users would only start with an USD balance and if they receive another type of currency, the system will create the balance for them with a fee.

- Type from this field we record the type of the transaction. It could be withdrawal, deposit, charge-fee or otherwise. In this design, we can easily charge a fee on any transaction in any way we want because there's only one entrance for all money movements. This simplifies implementation significantly.
- Value is the field used to record the amount of money involved in the transaction.
- Epoch is the actual epoch this transaction happened. It's partially ordered since in one epoch there could be many transactions, which would then be ordered by the order in which they happened.
- Description. This field allows system or users to add comments in their transaction.

Report
Reports are history of a certain user's transaction logs in a given period (e.g. 1 week, 1 month) It's a simple composition of a list of transaction objects, but for simplicity we encapsulate it in this object so that it's future proof.

Stock
Stock entities represent a certain amount of stock that has a deterministic owner. We maintain buyInPrice and current Price so that the realized and unrealized profits are easy to calculate. See complete business logic in the stockCtl class.

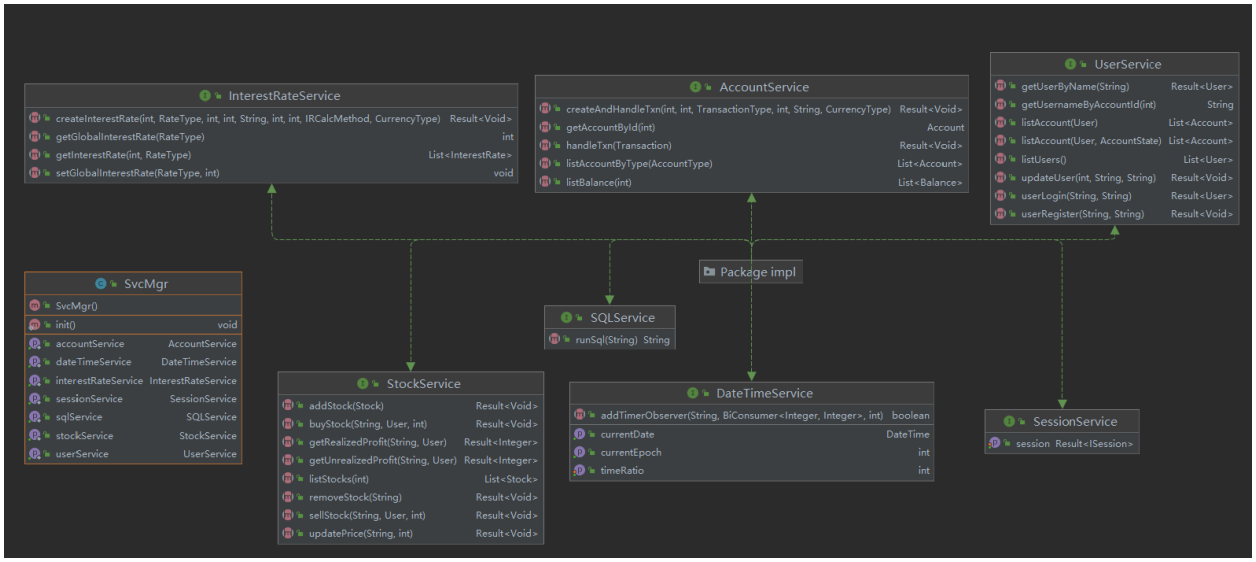
DateTime
DateTime is a critical part in our effort to correctly simulate time. Each datetime object corresponds to one virtual day, and the elapsedHours attribute indicates the actual hours. To ensure performance hours is our granularity of persistence. Time ratio is kept stable within a day for easy implementation. Only with a persisted time object can we get the exact same time upon reboot, even with different time ratios configured in the previous run. See complete logic in the DateTimeCtl class.

The factory package @Tingzhou

As its name suggests, this package contains the **abstract factories** used to create users, managers during account registration. We found that it benefits our code structure when we wanted to refactor the interest rate module and had to change the behavior of default users' account creation object.
The memory session is an unnecessary layer of abstraction for Web scenarios where there's a globalized data store storing which specific user is using the application at this time. Thanks to the MVC design of our project, it's extremely easy to add a web server for our backend, which handles RPC or HTTP calls. In that scenario, this memory session would actually make sense because requests will be Truly stateless, unlike our swing frontend, which is essentially single-user at a time.

The service package @Tingzhou

The service package has three parts, the interfaces, the implementations, and the Service Manager **singleton** object.



This is the critical part of the MVC design pattern, where external callers don't have to know the actual implementations to get things done.When the bank backend initiates, it initializes all the service controllers as **singletons** and registers these in the centralized **ServiceMangager** instance, which then serves frontend calls. We argue that this provides simplicity to the frontend as they no longer have to care about the specific underlying implementations and can focus on frontend logic. This also makes interface integration as well as refactoring relatively easy because everything's declared in the service interfaces, and all that frontend has to do is to ask for the specific controller from the ServiceManager and call whatever the function they want in the interface.

In a more advanced scenario, the service manager can be utilized in **aspect oriented programming** (AOP) and perform RBAC (Role based access control) for each of the declared interfaces through the use of dynamic proxies. However, that's way beyond the scope of this course and therefore we chose not to implement this part as it's not useful in a local environment anyways.

The interfaces are generally organized on the granularity of standalone function-sets, for example, the entire stock market is a standalone feature-set, allowing users to buy,sell stock and managers to add/remove stock as well as set/update stock price. We did not design a stock market with complicated stock orders simply because *we did not have time*.

As an effort to separate concern as well as implementing the best practice, the Entities do not have more business logic other than get/set, complicated business are instead implemented in the service controllers, where they operate on the static dao objects assigned to the entity classes at boot time.

Other Design patterns used @Tingzhou

The observer pattern

To centralize logging as well as having the ability to output to multiple log collectors (System.out, GUI), we have a standalone Logger that implements the observer pattern. Any observer can provide a callback function of type Consumer<String> to the logger singleton to receive

notifications on new logged events. Whenever the system produces a log the logger notifies all consumers in order.
The same observer pattern is also used in our Time Controller (see below for detailed design), where any object can register a callback function at the time controller so that after a certain interval the callback function is called and the object can decide its own behavior accordingly.

Technical perspective: controlling time @Tingzhou

To properly demonstrate time flow across possibly months over minutes in real life, we make an attempt to simulate time in this project. We introduce **epochs**, which map to real world hours. Our core challenges involve correctly implementing dynamically configurable time ratios in an event driven fashion, maintaining elapsed time across process reboots and losing as minimal amount of information as possible in case of catastrophic failure made possible by the transaction logs.

The thread model @Shu

We use thread to implement the timer in the program. To create an independent clock from the real world, we give the timer a unique time system so that it has a time conversion with the real world. To make sure the timer is running all the time, we implement the run method of the Thread class, and open a new thread to start it at the beginning of the program. When the program is running, this thread will be also running to make sure the time is changing in the program. In this situation, the loan interest can be charged and the deposit interest can be paid. There's a total of 3 threads in this project, the timer thread, the main backend thread, and the java swing's GUI event queue running on a separate thread.

Maintaining consistency @Tingzhou

A core issue with having both virtualized time and configurable time ratios with real life time is that we cannot peg on system clocks at all. Similar to tackling time differences in distributed systems, we use a similar notion like **Lampart's logical clock** in our project at the granularity of epochs. The latest time is kept at hour level in a persisted database, the DateTimeCtl instance serves a similar function as the TSO (timestamp oracle) in distributed databases, but only has hour-level accuracy. Transactions rely on both the epoch and logical ordering to be sorted. The DateTime entries act like WAL (Write ahead logs) and allow the system to tell how many epochs has actually passed since that day. Upon system reboot, the DateTimeCtl instance looks in the database for the latest DateTime entry and resumes from there. This ensures that datetime is continuous in our context.

However, this solution introduces another issue: there may be transactions that happened after the last epoch has happened but the next epoch hasn't been updated in the database. In this case, the transaction's epoch must be the latest epoch available. Upon reboot, this would cause data consistency issues among history transactions (Users seeing transactions that technically hasn't happened in our clock). We deal with this problem with the help of Transaction logs. By

rolling back the transaction logs one by one first thing upon DateTimeCtl reboot, we can guarantee that nothing is lost and no invalid data still persists.

Tolerating failures @Tingzhou

Since we utilize an ORM package to do the data access in the database, every operation is a transaction satisfying the ACID principle. This means that every operation is either successful or not, therefore guaranteeing that even if there had been OS failures, everything on disk remains safe.

Coding perspective: Eliminating Checked exceptions & null checks @Shu

Result<T> VS throws exception: where should errors be handled?

We add Result<T> to solve the exception problem. Through the Result class, we make most return values become result type. We can justify whether the return value is useful or null by checking the success parameter. If the result is successful, it means we will get the data properly and can pass it to the next function. If the result isn't successful, the return data may be null. Meanwhile, the result will include a message indicating the reason for failure, this will be printed in the log so we can check it easily.
Compared with throwing exceptions, **Result<T> can wrap errors and allow errors to propagate without manually declaring checked exceptions**. Our experience from this experiment is that this system works best with syntax sugars provided by newer, more modern languages such as Rust.

Contribution of project
Frontend Part: @Srinivas @Anoohya @Tingzhou Yuan @Shu Xing
Backend Part: @Tingzhou Yuan @Shu Xing

Conclusion

This project would have been so much fun if everyone had sufficient experience on this. We could have worked on things like front-end that listens to backend events and updates data autonomously, frontend data store, and explored stateful components w/ Swing that worked more like modern Javascript frameworks, or tried the old-school Swing way of storing entities in a database and use custom daos to fetch them. Instead we're stuck with this death march. Netbeans represents design philosophies in the previous decade, if not century, unfortunately. Nevertheless, thanks to the solid object design we had made initially, we were able to make most things work with relative ease. The transaction log concept eased our design and

implementation significantly. Despite time constraints, some of the interfaces that would reflect the true abilities of the underlying object designs are not declared, but our object design actually allows for even greater degrees of flexibility with regard to accounts, balances and interest rates.