Tal Derei

CSE 411: Assignment 2
Design Patterns

Design Patterns

CSE 341:

The insurance database application (written in *Java*) integrates the JBDC API for connecting to an Oracle database for data persistence. The database is encapsulated as a single object, and a single connection is being opened on that object (for each interface respectively). You can only connect to one interface at a time, and when that connection terminates, the resources will be returned to the database and the connection closed. Then that database object has multiple interfaces (i.e. different types of users) associated with it, and PreparedStatement are attached to that object as well. This “one class, one object” paradigm represents a **Creational Design Pattern (i.e. Singleton Pattern)**. The codebase implements a single connection object in ‘DBManager.java’, and passes that connection object to main, where it maintains a global reference to that connection object accessible by any other component in the application. Rather than creating multiple connections for different functionalities in the system that require a connection to the database for executing their SQL commands, they share access to this global connection object when a user logs in. This reduces the complexity of managing multiple connections for a single user (similar to a Pool design) every time they interact with the database.

```java
public class DBManager {
    private static Connection conn;

    /**
     * Establish a connection to the PostgreSQL Database
     * @return Connection object
     */
    public static Connection connect() {
        /* User-defined variables and database parameters */
        Connection conn;
        final String url = "jdbc:postgresql://ec2-52-1-20-236.compute-1.amazonaws.com:5432/dfocu1cnfk70bl";
        Scanner input = new Scanner(System.in);
        String username = "ezdvczoxegzrgf";
        String password = "e620a6a42fb3d6b0dd4d628325d441386eb62ccd53eee2439aa7b86c9caa91ed";

        /* try connecting to database with user-credentials */
        try {
            conn = DriverManager.getConnection(url, username, password);
            conn.setAutoCommit(false);
            System.out.println("Connected to the PostgreSQL database server successfully!");
        } catch (SQLException exception) {
            System.out.println("\nConnection to the postgres database server failed! Please try again!\n");
            return connect();
        }
        return conn;
    }
```

Figure 1: Connection Object Represented as “private static Connection conn”

CSE 297:

The custom blockchain written in C implements merkle-trees (data structure) for storing transactions in blocks, serialization of blocks between memory/disk, and merkle-proof validation for verifying the validity of blockchain transactions. We implemented a known **Creational Design Pattern (i.e. Factory Pattern)** for creating objects -- the creation logic for the object is abstracted from the client, instead referring to the object using a common interface. For example, we define an interface for creating a “block” object, and let subclasses decide which class to instantiate in order to verify the block based on the position of the blockchain. The reason why inheritance is necessary is because we need to differentiate between a genesis block (the first block) and subsequent blocks in the linked-list. And then serializing the blocks to disk and rebuilding them after reading them back into memory will be their own subclasses implementing the parent ‘block’ class that includes this factory method for instantiating blocks. Without this design pattern, there would be a lack of abstractions between the master “block” class, and the different block objects that can be partitioned at runtime depending on the location in the chain (i.e. is it a new blockchain, or has there been a hard-fork on the main chain).

```c
/* serialize_first_block serializes the first block from memory to disk (header contents and merkle tree) */
void serialize_first_blockchain(Block *block, FILE *write_blockchain){
    /* ***write data in BINARY representation to disk -- the data will look garbaled*** */
    Fwrite(block->header->previousHash, sizeof(unsigned char), 1, write_blockchain);
    Fwrite(block->header->rootHash, sizeof(unsigned char), 32, write_blockchain);
    Fwrite(&(block->header->timestamp), sizeof(unsigned int), 1, write_blockchain);
    Fwrite(&(block->header->target), sizeof(double), 1, write_blockchain);
    Fwrite(&(block->header->nonce), sizeof(unsigned int), 1, write_blockchain);
}
```

```c
/* serialize_blockchain serializes block[i] (not the first block) from memory to disk (header contents and merkle tree) */
void serialize_blockchain(Block *block, FILE *write_blockchain) {
    Fwrite(block->header->previousHash, sizeof(unsigned char), 32, write_blockchain);
    Fwrite(block->header->rootHash, sizeof(unsigned char), 32, write_blockchain);
    Fwrite(&(block->header->timestamp), sizeof(unsigned int), 1, write_blockchain);
    Fwrite(&(block->header->target), sizeof(double), 1, write_blockchain);
    Fwrite(&(block->header->nonce), sizeof(unsigned int), 1, write_blockchain);
}
```

Figure 2: Base Classes for Instantiating “Blocks” in the Blockchain