```java
/*
 * Hi!
 *
 * This DBMS was designed to allow a developer to swap out RecordFormat and
 * PageFormat. This allows someone to customize how the DBMS stores Records and
 * how Records are stored in each Page.
 *
 * This abstraction is done through the Strategy design pattern, where an
 * algorithm is abstracted out into another class to allow swapping at runtime.
 *
 * Everything else should be self explanatory, it is just very rigid pseudocode
 * (some things are missing) but I tried to be explicit as possible.
 *
 * Hopefully it's easy to read, I tried to add Java Docs to help with that.
 *
 * Let me know if you need any help!
 *
 * - Josh
 */

/**
 * Main interface to the DBMS.
 */
public class AccessManager implements AccessManagerInterface {
    /** Name of the database. */
    private String dbName;

    /** List of tables in the database. */
    private HashMap<String, Table> tables;

    /** The PageFormat to use when placing a Record into a Page. */
    private PageFormat pageFormat;

    /** The RecordFormat to use when writing a Record. */
    private RecordFormat recordFormat;

    /**
     * Creates a new AccessManager.
     * @param givenDbName Database info to access.
     * @param givenPageFormat PageFormat to access.
     * @param givenRecordFormat RecordFormat to access.
     */
    public AccessManager(final String givenDbName,
            final PageFormat givenPageFormat,
            final RecordFormat givenRecordFormat) {
        // Read in all the information from the first Page
        Page page = DiskManager.readPage(0);

        //
        // Here we would read in all the serialized information regarding the
        // database schema such as the list of tables in the database which
        // contains all the mapping information and so on...
        //

        // Assign constructor values
        dbName = givenDbName;
        pageFormat = givenPageFormat;
        recordFormat = givenRecordFormat;
    }

    /**
     * Creates a new Table.
     * @param tableName Name of the new table.
     * @param fields Array of Fields to use.
     */
    public final void createTable(final String tableName,
            final Field[] fields) {
        Table table = new Table(fields, pageFormat, recordFormat);

        tables.put(tableName, table);
    }

    /**
     * Checks whether a table exists in the database.
     * @param tableName Name of the table.
     * @return True or false depending on the existence of the table.
     */
    public final boolean tableExists(final String tableName) {
        return tables.contains(tableName);
    }

    /**
     * Deletes a table.
     * @param tableName Name of the table to delete.
     * @throws DBMSException on any error.
     */
    public final void deleteTable(final String tableName) throws DBMSException {
        Table table = getTable(tableName);

        table.delete();

        // Remove from the list of tables
        tables.remove(tableName);
    }

    /**
     * Reads and returns a record.
     * @param tableName Name of the table to search.
     * @param rid Record ID to read.
     * @return The record that matches the ID given.
     * @throws DBMSException on any error.
     */
    public final Record readRecord(final String tableName,
            final int rid) throws DBMSException {
        Table table = getTable(tableName);

        return table.readRecord(rid);
    }

    /**
     * Writes a given Record to its Table.
```

```java
     * @param tableName Name of the table.
     * @param rid ID of the Record.
     * @param record Record to write.
     * @throws DBMSException on any error.
     */
    public final void writeRecord(final String tableName,
            final int rid,
            final Record record) throws DBMSException {
        Table table = getTable(tableName);

        table.writeRecord(rid, record);
    }

    /**
     * Deletes a record.
     * @param tableName Name of the table.
     * @param rid Record ID to delete.
     * @throws DBMSException on any error.
     */
    public final void deleteRecord(final String tableName,
            final int rid) throws DBMSException {
        Table table = getTable(tableName);

        table.deleteRecord(rid);
    }

    /**
     * Sets the RecordFormat when serializing it.
     * @param format RecordFormat to use.
     */
    public final void setRecordFormat(final RecordFormat format) {
        recordFormat = format;
    }

    /**
     * Sets the PageFormat for when saving a Record to a Page.
     * @param format PageFormat to use.
     */
    public final void setPageFormat(final PageFormat format) {
        pageFormat = format;
    }

    /**
     * Closes the database to ensure everything is saved.
     */
    public final void close() {
        // Write all schema and mapping from records to pages to the first
        // Page.
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream serializer = new ObjectOutputStream(out);

        // Contains mapping from Database to Tables
        serializer.writeObject(db);

        // Contains mapping from Tables to Pages + Field info
        serializer.writeObject(tables);
        // Close serializer
        serializer.close();

        // Read the first Page into memory.
        Page page = DiskManager.readPage(0);

        // Write the Database information to it
        page.writeBytes(out.toByteArray());

        // Write it back to disk
        DiskManager.writePage(0, page);

        // Clean up
        out.close();
    }

    /**
     * Helper to check if a table exists and return it.
     * @param name Name of the table.
     * @return Table that matches.
     * @throws DBMSException on any error.
     */
    private Table getTable(final String name) throws DBMSException {
        Table table = tables.get(tableName);

        if (table == null) {
            throw DBMSException("Table " + tableName + " does not exist.");
        }

        return table;
    }
}
/**
 * Exception to throw in this DBMS implementation.
 */
public class DBMSException extends Exception {
    /**
     * Throw an Exception with the given message.
     * @param msg Message to throw.
     */
    @Override
    public DBMSException(final String msg) {
        super(msg);
    }
}
/**
 * Class that abstracts away the disk management operations in the DBMS.
 */
public static class DiskManager {
    /**
     * Creates and returns a new Page.
     * @return The new Page.
     */
    final Page createPage() {
        return null;
    }

    /**
     * Deletes a Page.
     * @param pid Page id to delete.
     */
    final void deletePage(final int pid) { }

    /**
     * Write a Page to disk.
     * @param pid Page ID to write.
     * @param page Page to write.
     */
    final void writePage(final int pid, final Page page) { }

    /**
     * Reads a Page from disk.
     * @param pid Page ID to read.
     * @return Page that was read.
     */
    final Page readPage(final int pid) {
        return null;
    }
}
/**
 * Field class for the DBMS.
 */
public class Field {
    /** Name of the field. */
    private String name;

    /** SQL Type of the field. */
    private String type;

    /** Optional variable size of the field. */
    private int variableSize;

    /**
     * Basic constructor for a Field.
     * @param givenName Name of the Field.
     * @param givenType Type of the Field.
     * @param givenSize Size of the Field.
     */
    public Field(final String givenName,
            final String givenType,
            final int givenSize) {
        // Assign constructor values
        name = givenName;
        type = givenType;
        size = givenSize;
    }

    /**
     * Getter for the Field name.
     * @return Name of the Field.
     */
    public final String getName() {
        return name;
    }

    /**
     * Getter for the Field type.
     * @return Type of the Field.
     */
    public final String getType() {
        return type;
    }

    /**
     * Getter for the size of the field.
     * @return Size of the field.
     */
    public final int getSize() {
        if ("int".equals(type)) {
            return 4;
        } else if ("varchar".equals(type)) {
            return variableSize;
        }
        // ..
        // continued...
        // ..
        else {
            // and so on...
            return 0;
        }
    }
}
/**
 * Implementation of RecordFormat for fixed record format.
 */
public class FixedRecordFormat implements RecordFormat {

    @Override
    public final byte[] writeRecord(final Record record) {
        int length = 0;

        // Find the length of all Fields
        for (Field field : record.getFields()) {
            length += field.getSize();
        }

        byte[] result = new byte[length];

        // Write all the Field values to the result byte array
        int i = 0, field = 0;
        for (Field field : record.getFields()) {
            // writeBytes(result, i, field.getSize(), record.getValue(field));
            field += 1;
            i += field.getSize();
        }

        return result;
    }

    @Override
    public final Record readRecord(final Field[] fields, final byte[] bytes) {
        Record result = new Record(fields);

        // Extract all the values from the given byte array into the Record
        int i = 0, field = 0;
        for (Field field : fields) {
            Object value;
            // value = readValue(i, field.getSize(), field.getType());
            i += field.getSize();
            record.setField(field, value);
        }

        return result;
    }
}
/**
 * PageFormat for packing in the Records.
 */
public class PackedPageFormat implements PageFormat {

    @Override
    public final byte[] writeRecords(final byte[][] recordBytes) {
        int length = 0;
        byte[] result;

        // Calculate the length of the entire Page
        for (int i = 0; i < recordBytes.length; i += 1) {
            length += recordBytes[i].length;
        }

        result = new byte[length];

        // Copy the bytes from the given Record bytes into the final byte array
        for (int i = 0, j = 0; i < recordBytes.length; i += 1) {
            // copyBytes(result, j, recordBytes[i]);
            j += recordBytes[i].length;
        }

        return result;
    }

    @Override
    public final byte[][] readRecords(final byte[] bytes,
            final int nRecords,
            final int recordLength) {
        byte[][] result = new byte[nRecords][recordLength];

        // Read the bytes from the given byte array into the corresponding
        // Record byte array
        for (int i = 0; i < recordBytes.length; i += 1) {
            // readBytes(result[i], bytes, i * recordLength, recordLength);
        }
        return result;
    }
}
/**
 * Page class for DBMS.
 */
public class Page {
    /** Page ID. */
    private int pid;

    /** Size of the Page. */
    private int pageSize;

    /** Page Format. */
    private PageFormat pageFormat;

    /** Record Format. */
    private RecordFormat recordFormat;

    /** Number of Records. */
    private int nRecords;

    /** Length of a Record. */
    private int recordLength;

    /** Given data. */
    private byte[] data;

    /** Records in the Page. */
    private Records[] records;

    /** Fields in each Record. */
    private Fields[] fields;

    /**
     * Creates a Page.
     * @param givenFields Fields of the Record.
     * @param givenData Data to initialize with.
     */
    public Page(final Field[] givenFields, final byte[] givenData) {
        fields = givenFields;
        data = givenData;

        byte[][] recordBytes = pageFormat.readRecords(data, nRecords, recordLength);

        records = new Records[recordBytes.length];
        for (int i = 0; i < recordBytes.length; i += 1) {
            Record record = recordFormat.readRecord(fields, recordBytes[i]);
            records[i] = record;
        }
    }

    /**
     * Set the Page ID.
     * @param givenPid New Page ID.
     */
```

```java
public final void setPageID(final int givenPid) {
    pid = givenPid;
}

/**
 * Getter for the Page ID.
 * @return The Page ID.
 */
public final int getPageID() {
    return pid;
}

/**
 * Sets the PageFormat.
 * @param format PageFormat to set.
 */
public final void setPageFormat(final PageFormat format) {
    pageFormat = format;
}

/**
 * Sets the RecordFormat.
 * @param format RecordFormat to set.
 */
public final void setRecordFormat(final RecordFormat format) {
    recordFormat = format;
}

/**
 * Return the number of records in the Page.
 * @return Number of records.
 */
public final int countRecords() {
    return nRecords;
}

/**
 * Returns if the Record is full or not.
 * @return True or False depending on full.
 */
public final boolean isFull() {
    int count = 0;

    for (int i = 0; i < recordBytes.length; i += 1) {
        count += recordBytes[i].length;
    }

    if (count + recordLength > pageSize) return false;
    else return true;
}

/**
 * Retrieve the given Record ID.
 * @param rid Record ID.
 * @return Record with the given Record ID.
 */
```

```
    public final Record retrieveRecord(final int rid) {
        for (Record record : records) {
            if (record.getRecordID() == rid) {
                return record;
            }
        }

        throw DBMSException("Table does not contain record " + rid);
    }
}
/**
 * Interface for arranging a series of Records on a Page.
 */
public interface PageFormat {
    /**
     * Writes an array of Record bytes to a Page.
     * @param recordBytes Record to write.
     * @return Array of bytes.
     */
    byte[] writeRecords(final byte[][] recordBytes);

    /**
     * Reads Record bytes from an array of bytes.
     * @param bytes Array of bytes to read.
     * @param nRecords Number of Records in the Page.
     * @param recordLength Length of each Record.
     * @return Array of Record bytes.
     */
    byte[][] readRecords(final byte[] bytes,
            final int nRecords,
            final int recordLength);
}
/**
 * Record class for the DBMS.
 */
public class Record {
    /** Record ID. */
    private int rid;

    /** All the Fields of a Record. */
    private Fields[] fields;

    /** Values of a Record. */
    private Objects[] values;

    /**
     * Get the Record ID.
     * @return Record ID.
     */
    public final int getRecordID() {
        return rid;
    }

    /**
     * Get all the Fields.
     * @return All the Fields.
```

```java
     */
    public final Fields[] getFields() {
        return fields;
    }

    /**
     * Get the Field at a given index.
     * @param i Index to get.
     * @return Field at the index.
     */
    public final Field getField(final int i) {
        return fields[i];
    }

    /**
     * Get the value at a given index.
     * @param i Index to get.
     * @return Value at the index.
     */
    public final Object getValue(final int i) {
        return values[i];
    }

    /**
     * Sets a field to a given value.
     * @param givenField Field to set.
     * @param value Value to set.
     */
    public final void setField(final Field givenField,
            final Object value) {
        // Find the corresponding field index
        for (int i = 0; i < fields.length; i += 1) {
            if (field.equals(givenField)) {
                break;
            }
        }

        // Set the corresponding value
        values[i] = value;
    }
}
/**
 * Interface for serializing a Record.
 */
public interface RecordFormat {
    /**
     * Writes a Record to a byte array.
     * @param record Record to write.
     * @return Array of bytes.
     */
    byte[] writeRecord(final Record record);

    /**
     * Reads a Record from an array of bytes.
     * @param fields Fields to look for.
     * @param bytes Array of bytes to read.
     * @return Record that was read.
     */
    Record readRecord(final Field[] fields, final byte[] bytes);
}
/**
 * Table class for DBMS.
 *
 * Abstracts away some of the Table management features from the AccessManager.
 */
public class Table {
    /** List of Fields in the Table. */
    private Fields[] fields;

    /** Mapping from Record ID to Page ID. */
    private HashMap<Integer, Integer> recordToPage;

    /** The PageFormat to use when placing a Record into a Page. */
    private PageFormat pageFormat;

    /** The RecordFormat to use when writing a Record. */
    private RecordFormat recordFormat;

    /** Array of all the full Page IDs. */
    private ArrayList<Integer> fullPages;

    /** Array of all the full Page IDs. */
    private ArrayList<Integer> nonFullPages;

    /**
     * Creates a new Table.
     * @param givenFields Fields to create.
     * @param givenPageFormat PageFormat to use.
     * @param givenRecordFormat RecordFormat to use.
     */
    public Table(final Fields[] givenFields,
            final PageFormat givenPageFormat,
            final RecordFormat givenRecordFormat) {
        // Set the values
        fields = givenFields;
        pageFormat = givenPageFormat;
        recordFormat = givenRecordFormat;
    }

    /**
     * Deletes all the Pages being used up by the Table.
     */
    public final void delete() {
        // Iterate over all the Pages, deleting them one by one
        for (Integer pid : recordToPage.values()) {
            DiskManager.deletePage(pid);
        }
    }

    /**
     * Reads a Record from the Table.
     * @param rid Record ID to read.
     * @return Record that was read.
     * @throws DBMSException on any error.
     */
    public final Record readRecord(final int rid) throws DBMSException {
        if (recordToPage.contains(rid)) {
            int pid = recordToPage.get(rid);

            Page page = DiskManager.readPage(pid);

            page.setPageFormat(pageFormat);
            page.setRecordFormat(recordFormat);

            return page.retrieveRecord(rid);
        }

        throw DBMSException("Table does not contain record " + rid);
    }

    /**
     * Writes a given Record to the Table.
     * @param rid Record ID to write.
     * @param record Record object to write.
     */
    public final void writeRecord(final int rid, final Record record) {
        Page page;

        if (recordToPage.contains(rid)) {
            int pid = recordToPage.get(rid);

            page = DiskManager.readPage(pid);

            page.setPageFormat(pageFormat);
            page.setRecordFormat(recordFormat);

            page.writeRecord(record);
        } else {
            // We need a Page with space
            page = nextAvailablePage();

            page.setPageFormat(pageFormat);
            page.setRecordFormat(recordFormat);

            // Save the Record to the Page
            page.writeRecord(record);

            // Store location of the Record
            recordToPage.put(rid, page.getPageID());
        }

        // Update our Heap of Pages
        if (page.isFull()) {
            nonFullPages.remove(page);
            fullPages.add(page);
        }
    }
```

```java
    /**
     * Deletes a Record from the Table.
     * @param rid Record ID to delete.
     */
    public final void deleteRecord(final int rid) {
        if (!recordToPage.contains(rid)) {
            throw DBMSException("Table does not contain record " + rid);
        }

        // Find out where the Record is
        int pid = recordToPage.get(rid);

        // Read the Page from disk
        Page page = DiskManager.readPage(pid);

        page.setPageFormat(pageFormat);
        page.setRecordFormat(recordFormat);

        // Delete the Record
        page.deleteRecord(rid);

        // Update our mapping
        recordToPage.remove(rid);
    }

    /**
     * Helper method to get the next available Page with space.
     * @return Page with space.
     */
    private Page nextAvailablePage() {
        for (Integer pid : nonFullPages) {
            Page page = DiskManager.readPage(pid);

            page.setPageFormat(pageFormat);
            page.setRecordFormat(recordFormat);

            if (!page.isFull()) {
                return page;
            }
        }

        throw DBMSException("Table is full.");
    }
}
```