REVERSE ENGINEERING

# H2 DATABASE
# RUNTIME TERROR

## Introduction

In the first section of the report, we have explained five different design patterns identified in the source code of the h2 database. In software engineering, a design pattern is a general solution to a commonly occurring or repeating software design problem. It provides a template that describes how to solve a problem. It can speed up the development process and improve the code readability.

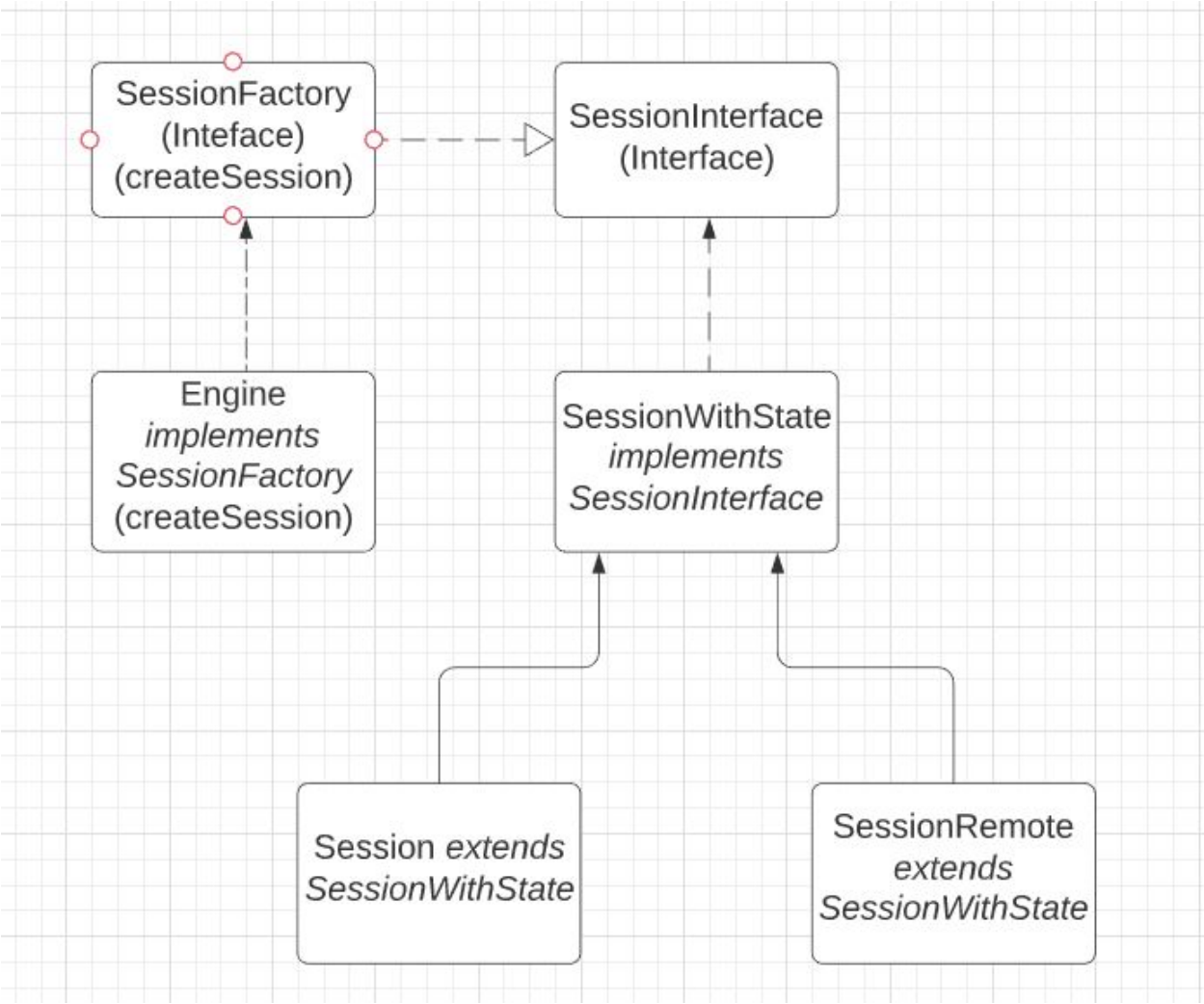
The second section consists of a brief description of the issue and the approach we used to fix the issue.

## Design Patterns

### Factory Pattern

**org.h2.engine.SessionFactory -** H2 database supports both embedded and Server(Remote session) modes which are implemented by using Session and SessionRemote classes respectively. Both these classes, in turn, extend SessionWithState class that implements the SessionInterface interface. Every time the user logs in to the database a new session is created by calling the createSession method defined in the SessionFactory interface. The implementation of this createSession method returns an object of SessionInterface type which can be either a Session or SessionRemote object. The benefit of using the Factory pattern here is that the classes implementing the SessionFactory's createSessionMethod can return a Session or SessionRemote object depending on the mode selected by the user.

Figure 1




## Singleton Pattern

**org.h2.engine.Engine -** The Engine class is implemented as a singleton class because the application should contain only a single instance of the Engine class that drives the whole system. The Engine class contains a map of all open databases and is responsible for opening and creating new databases.

Here an instance of the Engine class is created inside the Engine class as a static variable and can be accessed only using the Engine class's getInstance method. Also, the Engine class does not have a public constructor. Hence outside classes are prevented from

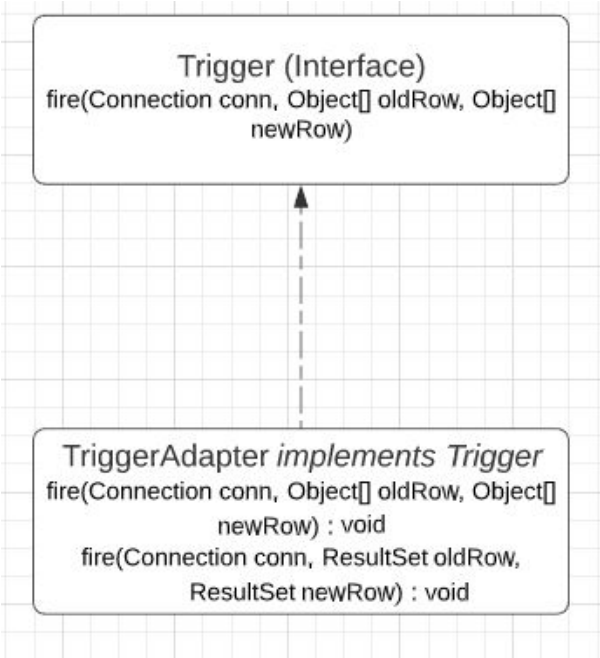
creating new instances of the Engine class and can only access the instance via the getInstance method.

```
private static final Engine INSTANCE = new Engine();
public static Engine getInstance() {
    return INSTANCE;
}
```

## Adapter Pattern

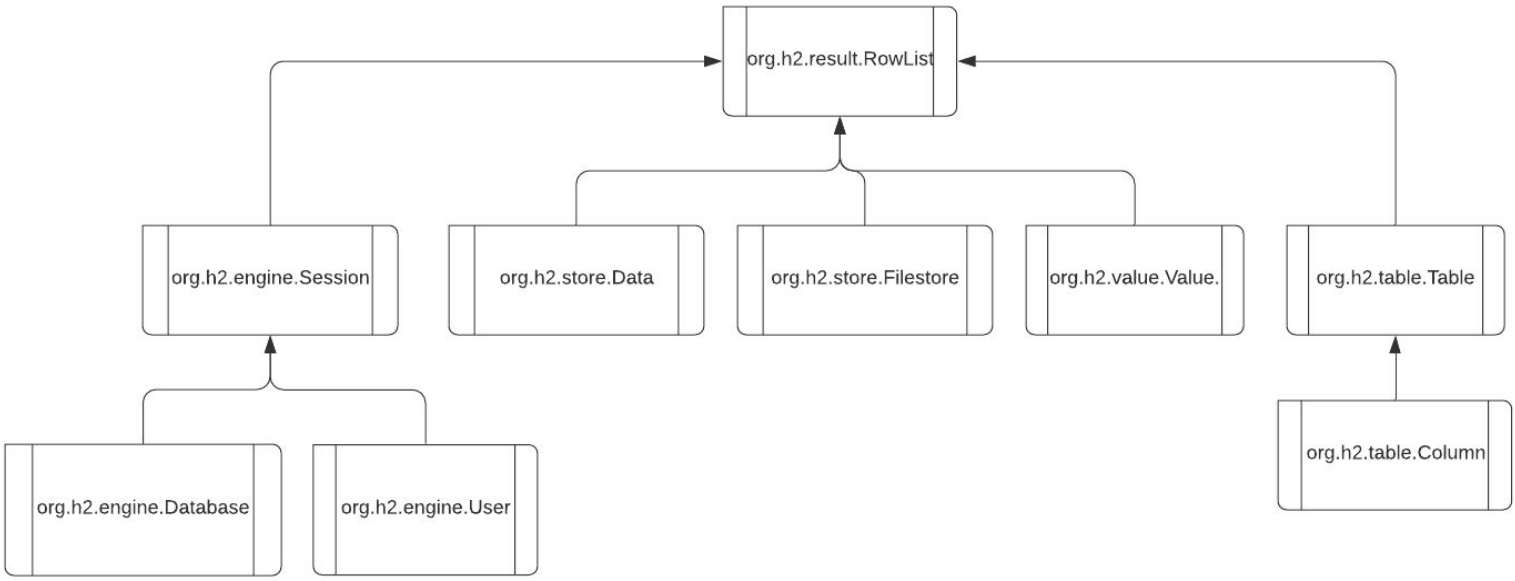
**org.h2.tools.TriggerAdapter -** TriggerAdapter is basically an adapter for the Trigger interface that allows using the object of type ResultSet instead of a row array. Trigger interface facilitates the initialization and firing of database triggers for INSERT, UPDATE, DELETE and SELECT statements. The fire method defined in the Trigger interface takes two Object arrays representing the old and new rows as the input parameters. On the other hand, the fire method implemented in the TriggerAdapter will take the Object Array that represents the rows and wraps it inside the ResultSet object. The benefit of this approach is that it allows other services working with Trigger objects to work with ResultSet type instead of row arrays.

Figure 2




## Decorator Pattern

Figure 3



**org.h2.result.RowList.java -** The RowList class extends its functionality by making use of additional classes such as Session, Data, FileStore, Value, Table, etc as illustrated by figure 3. Session class also extends itself by having Database and User objects as does Table class by using the Column class.

They have made use of the Decorator pattern by having each Decorator class hold a reference to the component it decorates in the form of an instance.

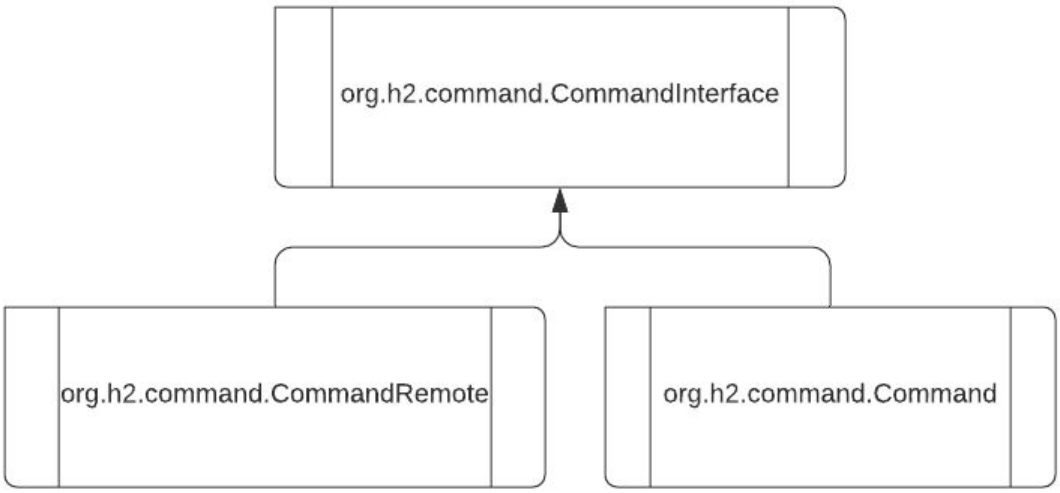
The decorator pattern allows us to extend the functionality at runtime. Also, another one of the main advantages is the pay-as-you-go approach it offers. Instead of adding all the features you need in a complex class, we just define a simple class and keep adding to it incrementally with the help of decorators.

## Command Pattern

**org.h2.command.CommandInterface -** Command, CommandRemote class are containers representing the SQL statement used in client-side and server-side respectively. Both the classes implement the interface CommandInterface. For each query to be processed, Session (Embedded Mode) and SessionRemote(Server Mode) class call the prepareCommand() function to create and return a CommandInterface object out of the SQL string query passed. Here, CommandInterface acts as a Command Class, which Command and CommandRemote implement it according to implementation for Embedded and Server modes respectively.

Using the command pattern allows us to encapsulate the request as an object, therefore making it easier to parametrize other objects with different requests. This advantage is demonstrated when Command and CommandRemote class have different implementations for the same functions as executeQuery(), stop(), close(), etc.

Figure 4




## Pull Request

We have submitted a pull request for the issue - https://github.com/h2database/h2database/issues/2360

Pull request - https://github.com/h2database/h2database/pull/2483(Note, we have created the fix, but the travis build seems to fail, the pull is not closed by the maintainers, and we are yet to figure out the cause of travis build failing).

The issue here is with the checkClustering method of org.h2.engine.Engine class. There is an if the condition that always returns false due to which the statement in the if block is never getting executed resulting in dead code.

Issue:

If clustering is enabled for the current database, we will first check if it is disabled for the current session. If it is disabled for the session, we then check if the strings clusterSession and clusterDb are not equal which will always be the case. Next, we are checking if database clustering is disabled or not. This will always return false since the database clustering was enabled in the first place (that's how we reached up to this point) and hence line number 268 will never get executed.

```
256 @     private static void checkClustering(ConnectionInfo ci, Database database) {
257           String clusterSession = ci.getProperty(SetTypes.CLUSTER,  defaultValue: null);
258           if (Constants.CLUSTERING_DISABLED.equals(clusterSession)) {
259               // in this case, no checking is made
260               // (so that a connection can be made to disable/change clustering)
261               return;
262           }
263           String clusterDb = database.getCluster();
264           if (!Constants.CLUSTERING_DISABLED.equals(clusterDb)) {
265               if (!Constants.CLUSTERING_ENABLED.equals(clusterSession)) {
266                   if (!Objects.equals(clusterSession, clusterDb)) {
267                       if (clusterDb.equals(Constants.CLUSTERING_DISABLED)) {
268                           throw DbException.get(
269                                   ErrorCode.CLUSTER_ERROR_DATABASE_RUNS_ALONE);
270                       }
271                       throw DbException.get(
272                               ErrorCode.CLUSTER_ERROR_DATABASE_RUNS_CLUSTERED_1,
273                               clusterDb);
274                   }
275               }
276           }
277       }
```

## Approach:

In order to fix the above deadcode issue, we added an else condition to the outermost if branch (Lines 275-280). So here when database clustering is disabled, we first check if clustering is enabled for the current session. If clustering is enabled for the session, then only we need to throw the “CLUSTER_ERROR_DATABASE_RUNS_ALONE” exception.

```
257 @     private static void checkClustering(ConnectionInfo ci, Database database) {
258           String clusterSession = ci.getProperty(SetTypes.CLUSTER,  defaultValue: null);
259           if (Constants.CLUSTERING_DISABLED.equals(clusterSession)) {
260               // in this case, no checking is made
261               // (so that a connection can be made to disable/change clustering)
262               return;
263           }
264           String clusterDb = database.getCluster();
265           if (!Constants.CLUSTERING_DISABLED.equals(clusterDb)) {
266               if (!Constants.CLUSTERING_ENABLED.equals(clusterSession)) {
267                   if (!Objects.equals(clusterSession, clusterDb)) {
268                       throw DbException.get(
269                               ErrorCode.CLUSTER_ERROR_DATABASE_RUNS_CLUSTERED_1,
270                               clusterDb);
271                   }
272               }
273           }
274           //Issue #2360 - Odd Condition: If cluster db is disabled but cluster session is enabled, throw an exception
275           else {
276               if(Constants.CLUSTERING_ENABLED.equals(clusterSession)){
277                   throw DbException.get(
278                           ErrorCode.CLUSTER_ERROR_DATABASE_RUNS_ALONE);
279               }
280           }
281       }
```