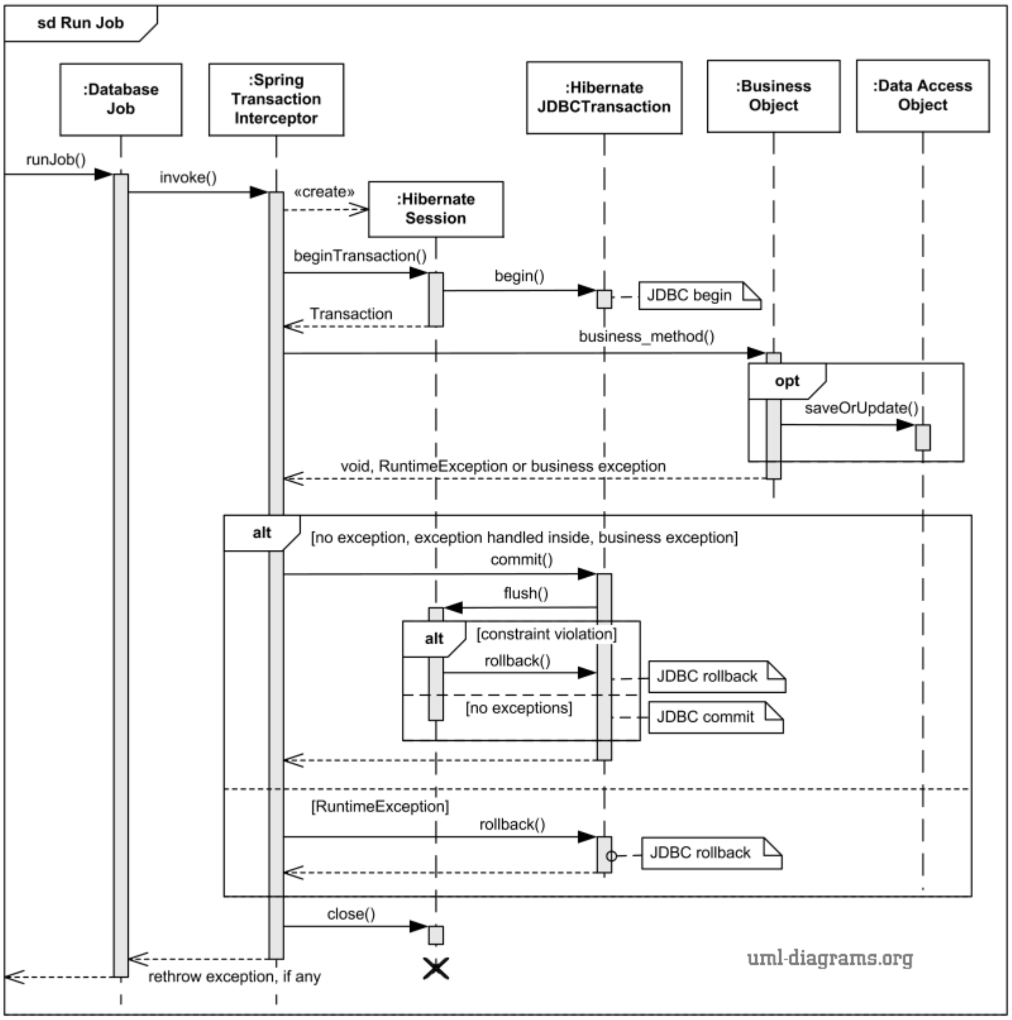
Design Pattern HW1
Marihebert Leal

1. **UML:**
   The following sequence diagram illustrates transaction management in relation to exception handling within the Spring application development framework.
   When some business method is called, it could be intercepted by Spring Transaction Interceptor. This behind the scene interceptor creates Hibernate session and starts Hibernate JDBC transaction, so that business method will run in the context of a new transaction.
   Business method execution could complete (successfully or not) without throwing any exception, or by throwing some Java runtime (unchecked) exception or some business (checked) exception.
   What happens if the business method throws some Java runtime (unchecked) exception?



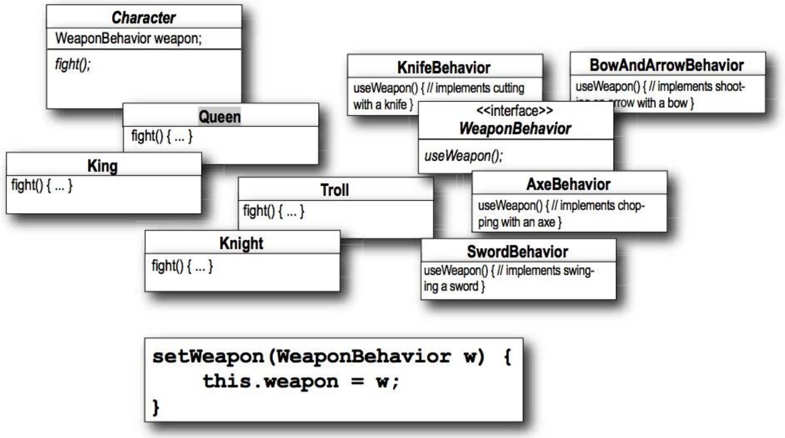
When the business method return a RuntimeException message to the Spring Transaction Interceptor Object, an alternative combination fragment take over. At this point in the sequence, if the return message is a RuntimeException, The Spring Transaction Interceptor Object sends a synchronous rollback() message to the Hibernate JDBCT Transaction object, which print a message “JDBCT rollback”. Since the rollback() is a synchronous message, it doesn’t send a return message until the rollback() is completed. After the rollback() is completed, a return message is sent to the Spring Transaction Interceptor Object and then a synchronous close() message is sent to the Hibernate Session object. The Hibernate Session Object is deleted by itself. Finally, The Spring Transaction Interceptor sends a rethrow exception message to The Database Job**.**

2. **Why can’t someone built a library of design patterns so we don’t have to write code every time we want to use them?**

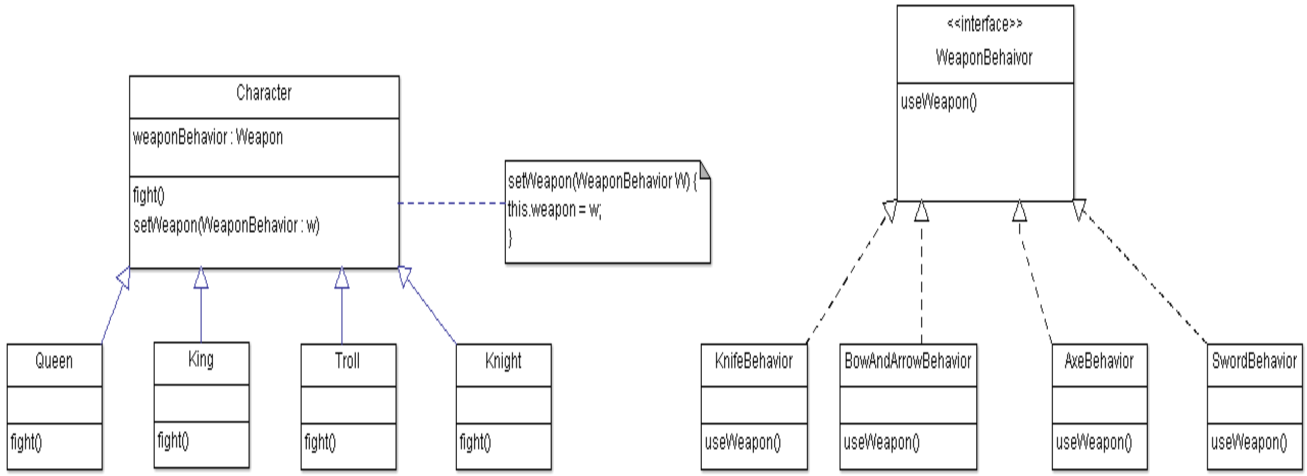
Design Patterns are higher level than libraries. Design Patterns let any programmer a better way to structure classes and objects to solve certain problems and it is the programmer job to adapt those designs to fit a particular application. Also, patterns are not something that you can put in a library and reuse for all the applications.

3. **Below you’ll find a mess of classes and interfaces for an action adventure game. You’ll find classes for game characters along with classes for weapon behaviors the characters can use in the game. Each character can make use of one weapon at a time, but can change weapons at any time during the game. Your job is to sort it all out... (Please use ArgoUML to draw the class diagram)**

1. **Arrange the classes. [2]**
2. **Identify one abstract class, one interface and eight classes. [1]**
3. **Draw arrows between classes. [2]**
4. **Put the method setWeapon() into the right class. [1]**



```
setWeapon(WeaponBehavior w) {
     this.weapon = w;
}
```



4. **In observer pattern, subjects “push” information to their observers. There is another way to update data - the observers may “pull” the information they need from the Subject. Please compare the “push” and “pull”. [2]**

Push means that the Subject will always push the data to observers, whether they need it or not.

Pull means that observers have the liberty of reaching for the data which makes sense for them, and can reduce on network traffic, and the application performance.

The difference between push and pull is just how much flexibility and liberty the observers have on the subject.

Push model

- All observers get entire new state when it changes.
- Observers need to do nothing.
- In the push model, information about the subject's state change is sent in the update message.
- Efficient: observer does need to determine what was updated.
- Requires the subject to know more about the observer (breaks abstraction).

- Observer always automatically receives the update, whether it wants it or not.

Pull model

- If different observers require different subsets of the state, they only pull what they need.
- If data interface changes, do not need to modify update method in each observer.
- Observers are responsible for acquiring the new state after an update() is called.
- Better transparency, subject doesn't need to know about observer.
- Observer is free to determine whether it wants to acquire the new state.
- Observer must determine what is new without help from the subject.