Concurrency

Optimistic Locking is supported to control the access to data at the same time. On the other hand, Pessimistic Locking that can be supported through Native API of Hibernate will be defined in JPA2.0 version.

Optimistic Locking

Without Locking Source

```
@Test
public void testUpdateUserWithoutOptimisticLocking() throws Exception {

    // 1. Enter a new data for testing
    newTransaction();
    addDepartmentUserAtOnce();
    closeTransaction();

    // 2. Inquire the user information using a same identifier
    newTransaction();
    User fstUser = (User) em.find(User.class,"User1");
    User scdUser = (User) em.find(User.class,"User1");
    closeTransaction();

    // 3. Change in Detached
    fstUser.setUserName("First: Kim");

    // 4. Change through separate transaction
    newTransaction();
    scdUser.setUserName("Second: Kim");
    closeTransaction();

    // 5. Changed by reflecting the work in 3.
    newTransaction();
    em.merge(fstUser);
    closeTransaction();
}
```

Let's examine the logic presented above in details.

1. Inquire the data using the same identifier in #1 and #2 codes
2. After the second transaction is terminated, #3 code changes userName of fstUser object in Detached status.
3. In the #4 code in the 3rd transaction, change userName of scdUser object. When the 3rd transaction is terminated, changes are reflected in DB
4. In the 4th transaction, perform update for fstUser object changed through #3 CODE.
5. Update fstUser or successfully complete

In conclusion, "userName of User that has userId of User1" becomes "First: Kim" and the update requested at scdUser before this was ignored. This phenomenon is called Lost Update and there are 3 methods to resolve this.

1. Last Commit Wins: If not performing Optimistic Locking, 2 transactions are successfully committed in basic type. Therefore, 2nd commit may overwrite the first commit contents. (in above example)
2. First Commit Wins: as the type applying Optimistic Locking, only first commit is successfully performed, an error is obtained at the second commit.
3. Merge: only 1st commit is successfully performed. An error is obtained at the second commit. However, different from First Commit Wins, do not start task for 2nd commit from the start again, but selectively change by selection of developer. It should be able to directly provide the method or screen that can merge the best strategy or changes.(additional implementation required)

JPA supports the execution of First Commit Wins through Versioning based Automatic Optimistic Locking. Add the Version to the relevant table to perform JPA Optimistic Locking. In this case, Version information is loaded when loading relevant table and mapped object, when updating the object, compare it with the current value of table to determine processing or not.

With Optimistic Locking Source

```
@Test
public void testUpdateDepartmentWithOptimisticLocking() throws Exception {

    // 1. Enter new data for test
    newTransaction();
    addDepartmentUserAtOnce();
    closeTransaction();

    // 2. Inquire Department information twice
    newTransaction();
    Department fstDepartment = (Department) em.find(Department.class,"Dept1");
    assertEquals("fail to check a version of department.", 0, fstDepartment.getVersion());
    Department scdDepartment = (Department) em.find(Department.class,"Dept1");
    closeTransaction();

    // 3. Set other deptName in Department information inquired and reflect in DB
    fstDepartment.setDeptName("First: Dept.");

    // 4. Call merge() method for the Department information inquired first
    newTransaction();
    scdDepartment.setDeptName("Second: Dept.");
    closeTransaction();

    // 5. Since DEPARTMENT_VERSION is already changed due to updating of 3rd transaction,
    //     StaleObjectStateException is expected to occur
    newTransaction();
    try {
        em.merge(fstDepartment);
        closeTransaction();
    } catch (Exception e) {
        e.printStackTrace();
        assertTrue("fail to throw StaleObjectStateException.",e instanceof StaleObjectStateException);
    }
}
```

As above, when performing the next testUpdateDepartmentWithOptimisticLocking() method, the 1st updating task is performed successfully, but StaleObjectStateException will be thrown like #6 code in the 2nd updating task. Following is the part of entity class setting for this.

Entity Class Source

```
@Entity
@Table(name="DEPARTMENT")
public class Department {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "DEPT_ID", length = 10)
    private String deptId;

    @Version
    @Column(name = "DEPT_VERSION")
    private int version;
...
```

```
}
```

The DEPT_VERSION can be added to manage versions to make optimistic locking.