

CS544
Enterprise Architecture
Midterm December 2016

Name _____

Student ID _____

NOTE: This material is private and confidential. It is the property of MUM and is not to be disseminated.

1. [10 points] **Circle** which of the following is TRUE/FALSE concerning Spring Inversion of Control/Dependency Injection:

T F Only Managed Beans can be injected in Spring, a POJO or JavaBean cannot.

EXPLAIN:_____ If a POJO or JavaBean can be a Spring Managed bean so they can be injected.

T F @Autowired works only on interfaces. It cannot work directly on classes.

EXPLAIN:___ It can work on classes. However you lose some of the value, testing; changing implementations

T F In practice, IoC container is not exactly the same as Dependency Injection as it involves a discovery step concerning the dependency.

EXPLAIN:___ IoC involves a “look up the dependency” step before injecting it. Spring has declarative configuration that identifies “where to find” the resource to inject.

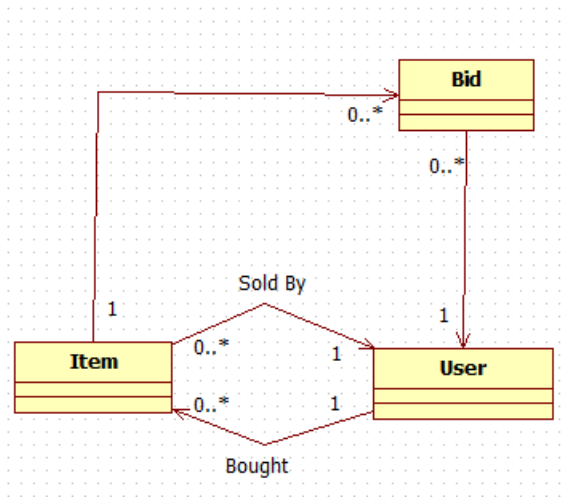
T F A domain object needed in a @Service class is usually a good candidate for Dependency Injection

EXPLAIN:_____ DI is mainly used for “cross” layer transitions, access plumbing resources, etc. NOT for passing business data around...

T F In Spring, DI can be done through either XML or through Annotations. They are mutually exclusive. That means, if you use XML for DI for one bean, they you should use it, exclusively for all beans.

EXPLAIN:_____ You could inject a DAO[memberDAO in memberService] through XML & inject another DAO[productDAO in productService] through @Autowired. Remember, XML configuration is the final word – if you configure the same bean twice, the XML configuration will take precedence.

2. [15 points] For the following relationships implement a SubSelect that fetches all items with their corresponding collection of bids.



What performance problem[s] does the SubSelect fetch address?

How does it work? – Explain the “algorithm” based on a universe of 10 Items each with a collection of 5-10 Bids.

Compare it to Join Fetch.

In Item.java

```
public class Item {  
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
    @Fetch(FetchMode.SUBSELECT)  
    private List<Bid> bids = new ArrayList<Bid>();  
}
```

In ItemServiceImpl.java

```
public List<Item> findbySubSelect() {  
    List<Item> items = (List<Item>)this.findAll();  
    // hydrate since LAZY load  
    items.get(0).getBids().get(0);  
    return items;  
}
```

Subselect does ONE fetch for the Items and ONE fetch for ALL collections. Therefore the number of Items & Bids in the collections does not matter,

It solves the Cartesian issue && the N+1 issue.

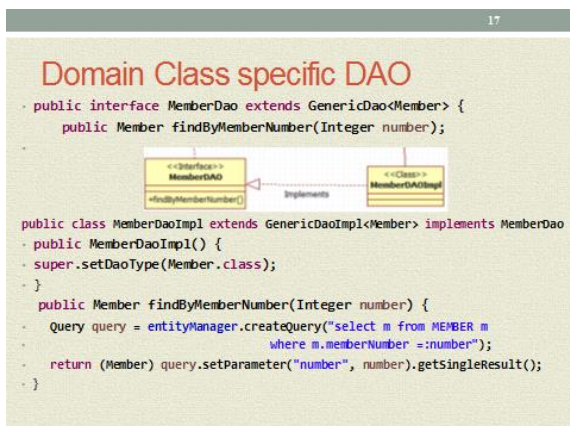
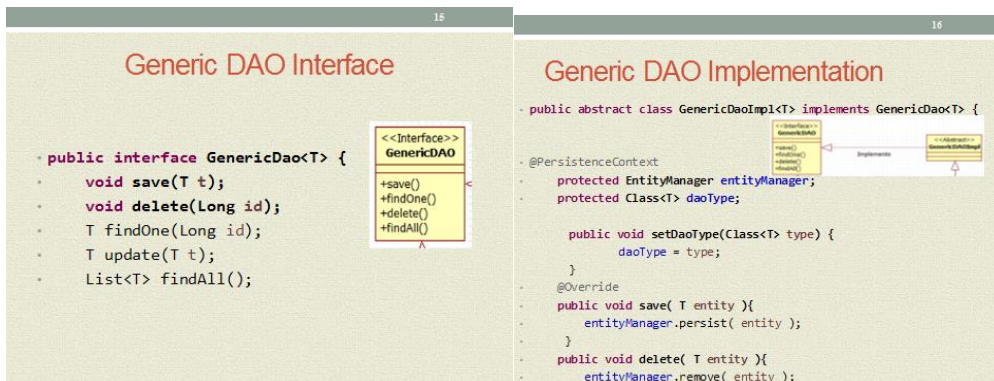
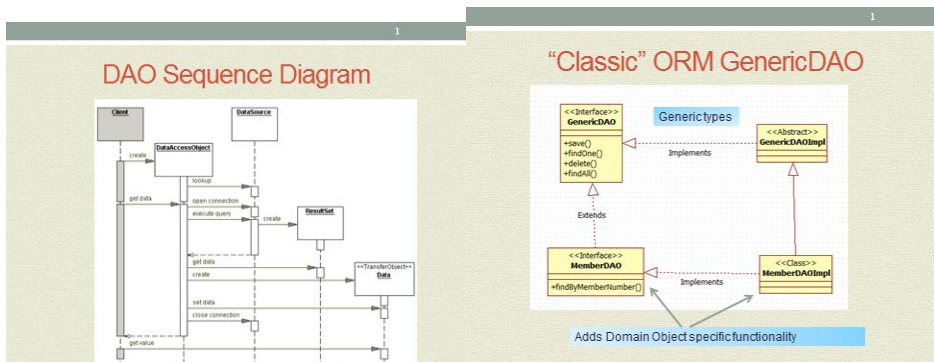
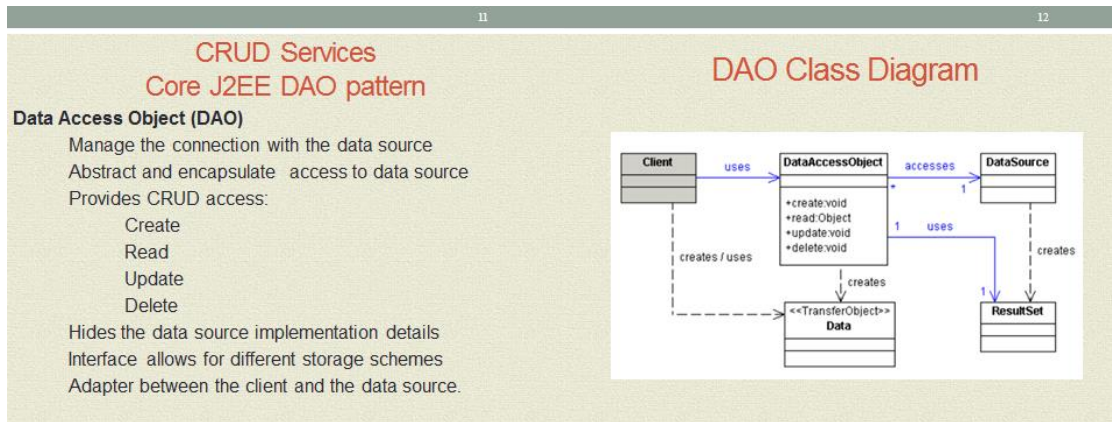
If the collection is LAZY loaded, then doing a get all for the “parent” entities [Item as in above] will get the “parent”/Item list. Within the same transaction, a reference to ONE of the Bid collections will fetch ALL the collections.

The Join Fetch, on the other hand will get ALL the Bids AND Items in ONE Select/fetch.

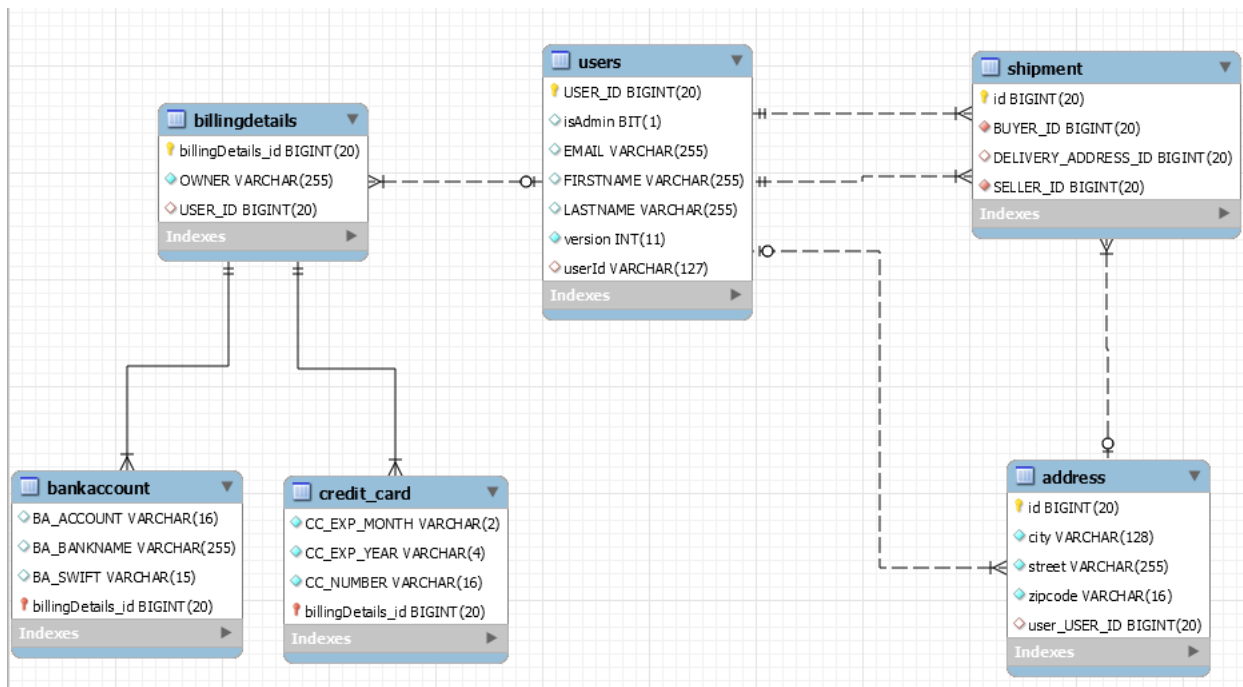
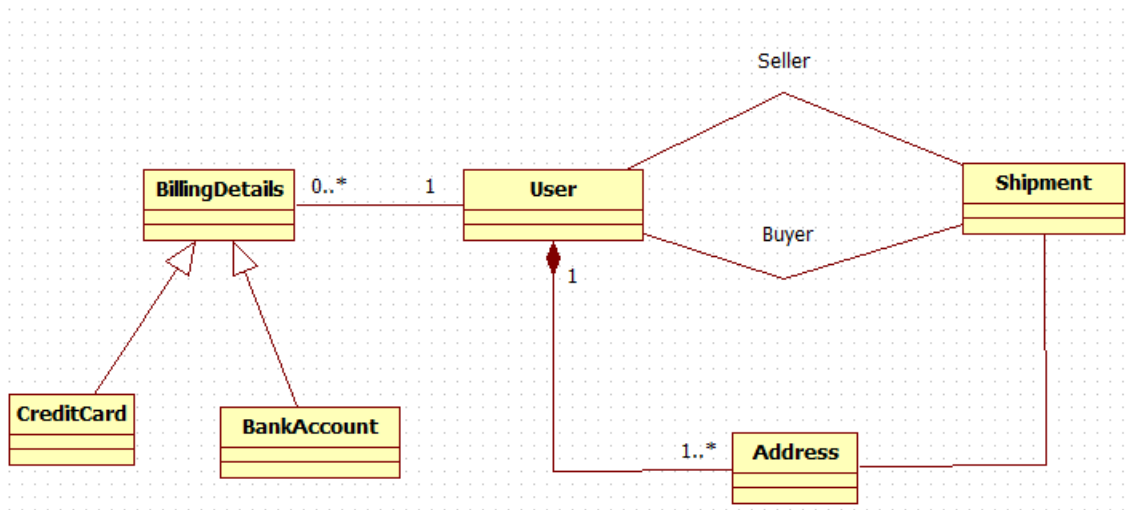
The Join Fetch, however suffers from the Cartesian product issue [ItemsXBids] which means that more than one copy of each Item/Bid pair will be included in the fetch. The number of copies depends on the number of Bids in the collection. So if Item A has 3 bids, 3 copies of Item A will be present.

4	5
<h3>Hibernate [Default] FETCH Strategy</h3> <p>Select fetching: a second SELECT [per parent N] is used to retrieve the associated collection. [DEFAULT] [N+1 Fetches] @Fetch (FetchMode.SELECT)</p> <ul style="list-style-type: none">• Join fetching: associated collections are retrieved in the same SELECT, using an OUTER JOIN. [1 Fetch] [EAGER] @Fetch (FetchMode.JOIN)• Subselect fetching: a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. [2 Fetches] @Fetch (FetchMode.SUBSELECT)• Batch fetching: Optimization of Select Fetching. Associated collections are fetched according to declared Batch Size (NBatch Size) + 1, @Batch Size(size=n)• @Fetch (FetchMode.SUBSELECT)• private Set<Address> addresses;	<h3>Hibernate Fetch Strategy Issues</h3> <ul style="list-style-type: none">• SubSelect depends on the "parent" query. If parent Query is complex, it could have performance impacts. If fetch=FetchType.LAZY need to "hydrate" children• BatchSize # of Fetches "unknown" UNLESS size of parent is constant Batch fetching is often called a blind-guess optimization If fetch=FetchType.LAZY need to "hydrate" children• Select N+1 TBA [To Be Avoided]• Join Cartesian – need to watch collection sizes; can be useful strategy

3. [15 points] The Core J2EE DAO pattern is fundamental to a well-organized ORM application. Explain the pattern, what is for, how it works. Include in the explanation, the sequence of interactions between the user, a DAO and the database. Include an explanation [& UML diagram] of the Generic DAO design. Be specific, give examples.



4. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the Domain Objects are not listed. All the fields are not listed. Only annotate the objects and fields that are listed.



```

22 @Entity
23 @Table(name = "USERS")
24 public class User implements Serializable {
25
26     @Id @GeneratedValue(strategy=GenerationType.AUTO)
27     @Column(name = "USER_ID")
28     private Long id = null;
29
30     @Column(name = "FIRSTNAME")
31     private String firstName;
32
33     @Column(name = "LASTNAME")
34     private String lastName;
35
36     @Column(name = "EMAIL")
37     private String email;
38
39     @Column(name = "isAdmin")
40     private boolean admin = false;
41
42     @OneToOne(fetch=FetchType.EAGER, cascade = CascadeType.ALL)
43     @JoinColumn(name="userId")
44     private UserCredentials userCredentials;
45
46     @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
47     private Set<Address> addresses = new HashSet<Address>();
48
49     @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="buyer")
50     private Set<Shipment> buyShipments = new HashSet<Shipment>();
51
52     @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="seller")
53     private Set<Shipment> sellShipments = new HashSet<Shipment>();
54
55     @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
56     private Set<BillingDetails> billingDetails = new HashSet<BillingDetails>();
57
58 }

```

```

6 @Entity
7 @Table(name = "SHIPMENT")
8 public class Shipment {
9
10     @Id @GeneratedValue
11     private Long id = null;
12
13     @ManyToOne(fetch = FetchType.EAGER)
14     @JoinColumn(name="DELIVERY_ADDRESS_ID")
15     private Address deliveryAddress;
16
17     @ManyToOne(fetch = FetchType.EAGER)
18     @JoinColumn(name="BUYER_ID")
19     private User buyer;
20
21     @ManyToOne(fetch = FetchType.EAGER)
22     @JoinColumn(name="SELLER_ID")
23     private User seller;

```

```

25 @Entity
26 public class Address implements Serializable {
27
28     @Id
29     @GeneratedValue(strategy=GenerationType.AUTO)
30     private Long id = null;
31
32     @Column(length = 255)
33     private String street;
34
35     @Column(length = 16)
36     private String zipcode;
37
38     @Column(length = 128)
39     private String city;
40
41     @ManyToOne(fetch=FetchType.LAZY)
42     private User user;
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```


5. [10 points] Explain the concept of locking. Include the definition of the two strategies covered in class. Give the details of Version-Based Optimistic Concurrency [Locking], how it relates to isolation levels, how it is implemented in JPA.

Lock Mode for Data Consistency

Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Locking strategies

- Optimistic Lock**
 - Concurrent transactions can complete without affecting each other, Transactions do not need to lock the data resources that they affect.
- Pessimistic Lock**
 - Concurrent transactions will conflict with each other
 - Transactions require that data is locked when read and unlocked at commit.

Version-Based Optimistic Concurrency[Locking]

- High-volume systems
- No Connection maintained to the Database [Detached Objects]
- Effective in **Read-Often Write-Sometimes** scenario
- Uses **read committed isolation level**
 - Guarantees **repeatable read isolation level**
 - An exception is thrown if a conflict occurs
- @Version Long version;**
 - When entity is updated - version field is incremented.

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
 - Keeps version #s so that it knows when they do
 - Guarantees best performance and scalability
 - The default way to deal with concurrency
 - There is no locking anywhere
 - It works well with very long conversations, including those that span multiple transactions
- First commit wins instead of last commit wins
 - An exception is thrown if a conflict would occur
 - ObjectOptimisticLockingFailureException

Optimistic Locking [Cont.]

- It works well with long conversations, including those that span multiple transactions
- LONG CONVERSATION Use Case:**
 - A multi-step dialog, for example a wizard dialog interacts with the user in several request/response cycles.
- session-per-request-with-detached-objects**
 - @Version Locking manages
 - Detached object consistency

JPA Optimistic Locking

JPA version-based concurrency control

- Simply add @Version to an Integer field – JPA takes care of the rest
- @Id
- @GeneratedValue(strategy = GenerationType.AUTO)
- private Long id = null;
- @Version

@Version property is incremented on every DB write
- @Column(name = "version")

Checked for consistency on every update
- private int version = 0;
- Hibernate:**
 - update purchaseOrder
 - set orderNumber=?,version=?
 - where id=? and version=?

6. [15 points] Implement a JQPL query that looks up a User by email who bought an Item with a shipping address that has a specific zip code. [Reference UML in problem #4]
For instance:

Find User who has an email address of JohnDoe@mail.com who bought an Item that was shipped to a zip code equal to 52556

OR

Find User who has an email address of JBean@post.com who bought an Item that was shipped to a zip code equal to 12345.

The Query should be a parameterized query. Also identify all the classes in the specific packages that need to be modified to adhere to the N-Tier architecture convention.

ANSWER:

edu.mum.dao. UserDao

```
public User findByBoughtItemShippedZip(String email,String zipCode);
```

edu.mum.dao.impl. UserDaoImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {  
  
    Query query = entityManager.createQuery("select u from User u,Shipment s  
        where u.email =:email and s.buyer = u "  
        + " and s.deliveryAddress.zipcode = :zipCode");  
    return (User) query.setParameter("email", email)  
        .setParameter("zipCode", zipCode).getSingleResult();  
}
```

edu.mum.service. UserService

```
public User findByBoughtItemShippedZip(String email,String zipCode);
```

edu.mum.service.impl. UserServiceImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {  
    return userDao. findByBoughtItemShippedZip (email,zipCode);  
}
```