CS392F: Automatic Software Design

P5: Feature House Assignment

Xiaohui Chen

xhchen0328@utexas.edu

Jianyu Huang

jianyu@cs.utexas.edu

Please unzip the three 7-zip files to have three folders. Please import them to eclipse to enable featureIDE

Running instructions (bash)

Before you run the script, please include hamcrest-all-1.3.jar, junit-4.11.jar, RegTest.jar to your CLASSPATH environment variable

Please use "bash run.script.sh" to run the bash script

The bash script runs TestWrapper class in each of the three parts. TestWrapper class uses JUnitCore.runClasses method to run all the JUnit test cases (detail descried later). Then the wrapper calls the Main class, which contains a test case of Entity arrays. The standard output would be redirected to a file called "out.txt" in each P5_Part* folder. The Utility class will then compare out.txt with expected*.txt, which contains the expected results. If the two files are the same, the wrapper will exit normally. Otherwise it will throw an exception.

Note that in part 1 and part3 the Main class has two different outputs. One output corresponds with the existence of remove feature and the other does not. Therefore the expected output without remove feature is stored in expected12.txt and the one with remove feature is stored in expected2.txt.

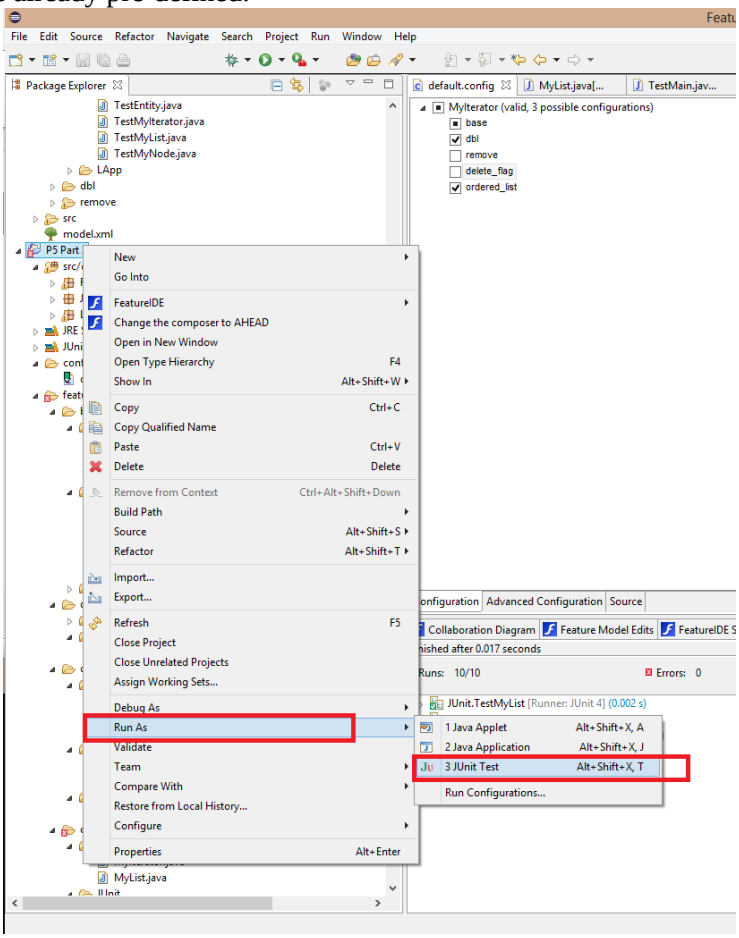
In part2, expected1.txt stores the expected result without remove, delete_flag and ordered_list features, expected2.txt stores the one with remove or delete_flag but not ordered_list, expected3.txt stores the one with ordered_list but no delete_flag or remove, and expected4.txt includes all features (note that remove and delete_flag is mutually exclusive but either feature has the same output)

Running instructions(Eclipse)

Please import the following projects into EcliseFeatureIDE, corresponding to Part 1, Part 2, Part 3 for each folder.

P5_Part1 P5_Part2 P5_Part3

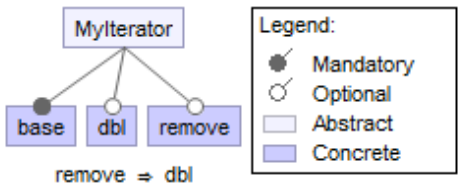
After modifying “default.config” to add the features, you can run batch JUnit Tests with JUnit testing scripts which we have already pre-defined.



Part 1: Creating a FH SPL

Description

The feature model of linked list is shown as follows:



The base feature is a singly linked list (class MyNode only has right pointer) while dbl feature adds the left pointer in MyNode class to make it double linked list. The remove feature adds the remove method in MyIterator class and delete method in MyList class. Also note that there is a constraint “$remove \Rightarrow dbl$”, which means the selection of remove feature means the selection of double linked list.

In the base feature, the remove method in MyIterator is an empty method. This is because the MyIterator class implements Iterator interface and remove method is required to be implemented. Here we just leave it blank.

There are some helper methods which do not exist in the original code given, such as getHead() method in MyNode class. Those helper methods are for JUnit testing, which will be described later. There will also be helper methods in all features and all parts of this assignment to aid testing.

The dbl feature adds the global variable left of type MyNode in class MyNode. It serves as the left pointer. Also insert method in MyList class is modified to let left pointer correctly set up during insertion.

Finally the remove feature modify the remove method in MyIterator and add a method called delete in MyList to support linked list deletion.

In part 1, Entity class serves as the Object to be in the linked list. It is indeed a pair of name and age. There are also two static arrays of Entity for usage in Main class. The Main class first inserts array one to the linked list. After printing out array one, it further inserts array two into the linked list. Note that without remove feature, the removal of the second array from the linked list is not executed since there is no remove functionality available.

Therefore the correct output for Main method when remove feature is selected is:

```
original list
(Chili, 20)
(Beth, 22)
(Scarlett, 7)
(Chief, 3)
(Steve, 90)
(Don, 60)

augmented list
(Kelsey, 25)
(Haggis, 1)
(Chili, 20)
(Beth, 22)
(Scarlett, 7)
(Chief, 3)
(Steve, 90)
(Don, 60)
```

The correct output for Main class when remove feature is selected is:

```
original list
(Chili, 20)
(Beth, 22)
(Scarlett, 7)
(Chief, 3)
(Steve, 90)
(Don, 60)

augmented list
(Kelsey, 25)
(Haggis, 1)
(Chili, 20)
(Beth, 22)
(Scarlett, 7)
(Chief, 3)
(Steve, 90)
(Don, 60)

revised list
(Chili, 20)
(Beth, 22)
(Scarlett, 7)
(Chief, 3)
(Steve, 90)
(Don, 60)
```

This matches the expected output.

JUnit Test

The Possible Configurations and JUnit test result are as the followings:

base	dbl	remove	Junit
√			Pass
√	√		Pass
√	√	√	Pass

The JUnit test files are stored in JUnit folder in each feature. The JUnit tests mainly test the return values by the methods in MyNode, MyList, Entity and MyIterator classes. Because Main class itself is a testing case, there is no JUnit test for the Main class.

We mainly use three Entity Objects for testing: (Someone, 50), (Somebody, 100) and (ThirdParty, 80). Through insertions, deletions and manipulations (e.g check the left/right pointer of a certain node), we compare the outputs with the expected ones. In fact, the JUnit tests pass without any error. Note that the JUnit tests are slightly different because the JUnit code is modified by FeatureIDE according to the features selected.

Part 2: Extending your FH SPL

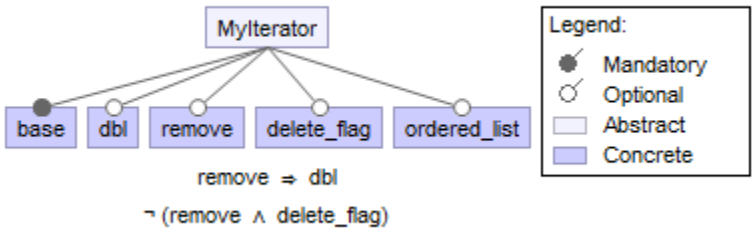
Description

Add two more features to Part 1:

Delete Flag – simply mark list elements deleted, and don't reclaim (remove) deleted nodes.

Ordered List – elements on a list are maintained in key order. This implies that the Entity class will need to support the Comparator<T> interface.

The feature model of linked list after adding two more features is shown as follows:



We add a new constraint here: "$\neg(remove \wedge delete\_flag)$", because we cannot delete the nodes physically and virtually at the same time.

Implementation details for Delete Flag

We add a boolean value, *delete_flag*(with default value *false*) as member in MyNode class, which will be used to mark whether the list element is deleted or not. If *delete_flag* is *true*, then the list element is marked as deleted.

Now when we want to delete a node in the linked list, we only need to mark this value as true. However, if the node is the head node, we also need to update its right node as the head. (See *MyList.java*)

Furthermore, we need to update *hasNext(), next(), remove()* function in MyIterator class. If we want to iterate the linked list, we need to skip those nodes which has already been marked as deleted. If we want to find the next node in the iteration, we only need to find the first node in the right of the current node which is not marked deleted. In the implementation of these iteration functions, we take care of both single linked list and double linked list cases, so that feature *Delete Flag* can be compatible with the case with dbl or without dbl.

Implementation details for Ordered List

The main change is that we modify *insert()* function in MyList class. Instead of just adding the node before the head node and updating the current as the head node, we iterate the linked list to find the first node whose value is larger than the key value of the new node. Then we insert this node before the node we just find. We take care of all possible corner cases. If the key value of the new node is smaller than the key value of the first node, we need to insert the new node before the head node and update the head node; if the key value of the new node is larger than the key value of the last node, we need to insert the node after the last node and update the tail node. We need to record the preNode so that we can insert the new

node for the case of single linked list. On the other hand, we add a helper function "*addLeftLink*()" to insert the new node when we enable the double linked list feature, because we not only need to update the right link we also need to update the left link to keep the list double linked.

To compare the key value of two nodes, we use the Comparator<T> to encapsulate the Entity class as a EntityComparator class. When we want to compare the key value of two nodes, we only need to declare a newEntityComparator class, and use its compare function. Here we use the String name as the key for the Entity, as Dr. Batory pointed out in Piazza.

JUnit Testing

The are totally 10 possible configuration combinations after we add two more feaures. The Possible Configurations and JUnit test result are as the followings:

base	Dbl	remove	delete_flag	ordered_list	Junit
√					Pass
√	√				Pass
√	√	√			Pass
√			√		Pass
√	√		√		Pass
√				√	Pass
√	√			√	Pass
√	√	√		√	Pass
√			√	√	Pass
√	√		√	√	Pass

We consider test cases for all possible configurations. The most fancy part is that our JUnit testing scripts are also been generated by FeatureIDE/Feature House.

Part 3: Creating a Generics-based SPL

Description

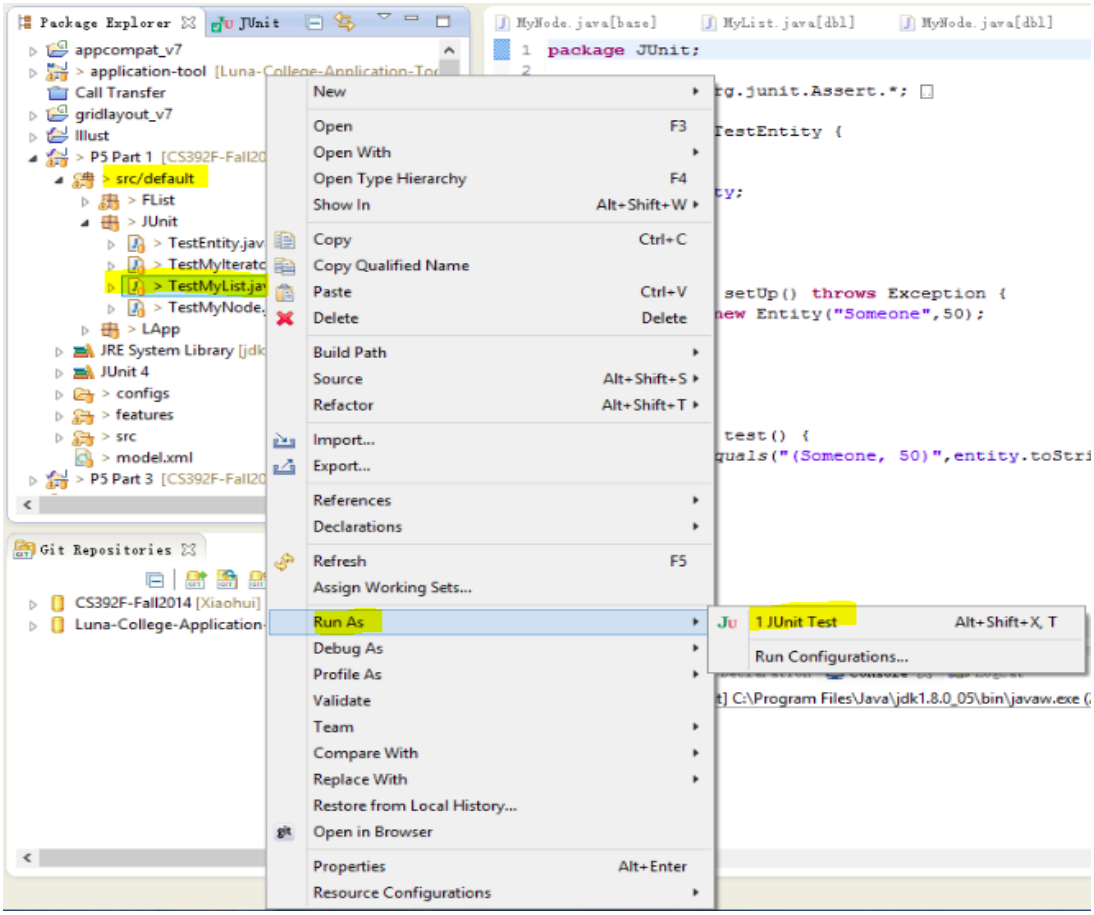
The feature model in part 3 is the same as that of part 1. However, in the implementations, classes MyNode, MyList and MyIterator all support generic type <T>, in all features. Therefore, in the Main class, those objects have generic type <Entity>. The rest of the implementation is the same as that of part 1. The output of Main class is the same as that of part 1. Therefore, the output matches the expectation.

JUnit Testing

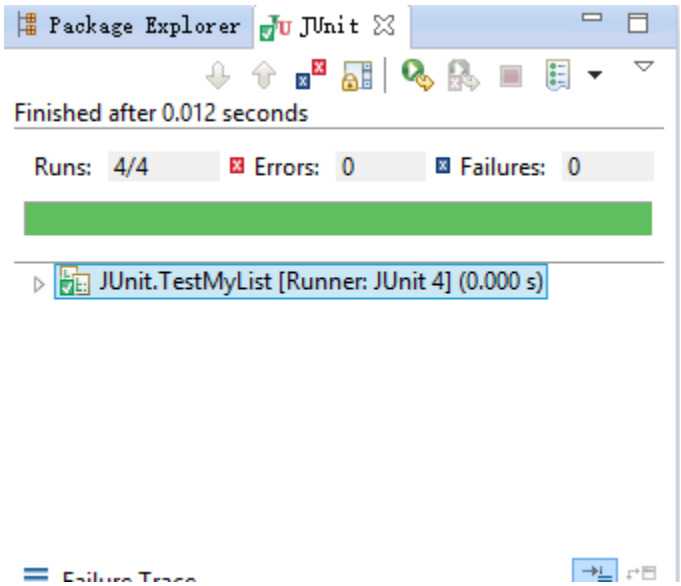
Instead of using Entity objects, we use Integer 80,50, and 100 for the JUnit testing. Undoubtedly, the MyNode, MyList and MyIterator all have generic type <Integer>. The rest of the implementation is the same as that of part 1. In fact all the JUnit tests passed and the program which support generic type linked list is correct.

Miscellaneous

To run a single JUnit test after selecting the deisired features, go to src/JUnit and find the desired JUnit testing case. Then right click the file and run as JUnit Test, as shown below.



The example output of JUnit Test is shown as follows.



The green bar means all the test cases in the JUnit class file have passed. In fact, our program passes all the JUnit tests.