McGill UNIVERSITY

ECSE 429 Software Validation
Fall 2017
Project Part 2

Group 10

Michael MAATOUK	260554267
Lucien GEORGE	260571775
Alexander BRATYSHKIN	260684228
Nathan LAFRANCE-BERGER	260686487
Benjamin WILLMS	260690005

Table of Content

Introduction:

As we began working on the second deliverable, we quickly came to realize one very important characteristic of our dependency graph: it is bound to be disjoint. This is due to the fact that we are restrained solely by the 20 classes from the Apache Commons Collection specified in the instructions. Naturally, we could've found concrete implementations of some classes that we were missing, or could've joined the graph due to the object-oriented nature of Java (after all, all classes inherit from Object), but since these relationships fall outside the scope of the deliverable, we omitted them.

To facilitate the grader's jobs, we decided to split the report into different sections. Each section corresponds to a cluster of related classes within our dependency graph. Since every operation, table, and justification is "local" to the cluster (e.g. changes in Closure, SwitchClosure, CatchAndRethrowClosure and ChainedClosure are not going to affect any other classes outside of their own family), it felt simpler for us (and hopefully for the grader as well), to split everything accordingly. At the end, you will find a complete and holistic overview of all required elements of the report, including the final object relationship diagram, the class firewall table, two testing level tables, the final testing order graph and the description of our drivers and mocks.

Additionally, it is indispensable for us to mention that we were heavily inspired by the research paper Prof. McIntosh provided us with entitled **"Testing Levels for Object-Oriented Software"** by Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.H. Durand. We drew practically all of our justifications and methodology from the publication, and we referenced the pages in our justifications accordingly. The first section, Cluster Family, goes more into detail into the explanation of particular terminology we've used and elaborates on the idiosyncrasies of Labiche et al.'s extension to the Kung et al. approach which, as we all know, presents certain shortcomings since it does not consider dynamic relationships between classes, nor does it give "explicit information on the other classes that have to be involved in the test experiments" *(p. 138)*, and also neglects the presence of abstract classes.

Closure Family

The “Classes of Closure Family” tag which recurs throughout this section refers to the set of dependencies between the following classes: Closure, SwitchClosure, ChainedClosure, CatchAndRethrowClosure.

We begin by analyzing the source code for the Apache Commons Collection. Looking at Closure, we notice that it is an interface. Further inspection of the codebase reveals that SwitchClosure, ChainedClosure and CatchAndRethrowClosure all implement said interface. CatchAndRethrowClosure, itself, is also an abstract class, with no concrete implementation provided (we will develop further on this towards the end of this section). Within ChainedClosure, we observe that there is a dependency injection which is highlighted through the operations within the class’ private constructor:

```
private final Closure<? super E>[] iClosures;
...
private ChainedClosure(final boolean clone, final Closure<? super E>... closures) {
        super();
        iClosures = clone ? FunctorUtils.copy(closures) : closures;
}
```

This implies that there is an aggregation of objects of type Closure. Therefore, ChainedClosure both aggregates and inherits from Closure. Likewise, if we dive into the SwitchedClosure, we can extract an identical relationship to class Closure from its own dependency injection:

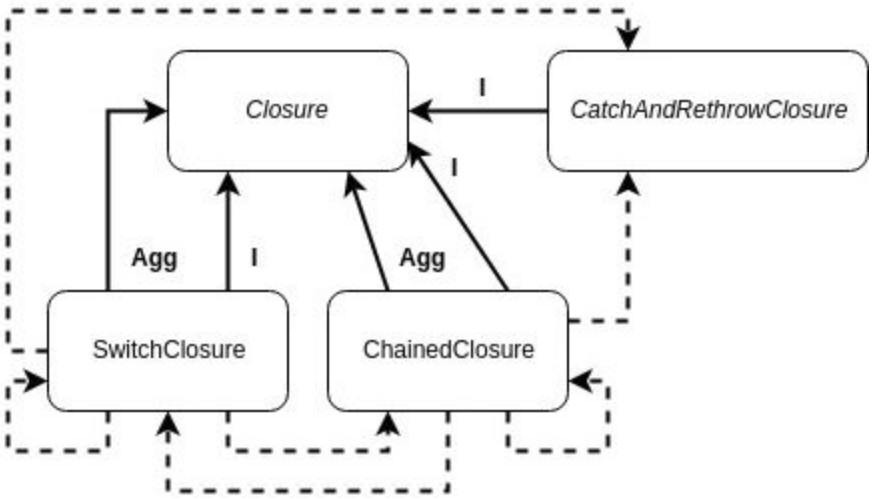
```
private final Closure<? super E>[] iClosures;
private final Closure<? super E> iDefault;
...
private SwitchClosure(final boolean clone, final Predicate<? super E>[] predicates,
                      final Closure<? super E>[] closures, final Closure<? super E> defaultClosure) {
        super();
        iPredicates = clone ? FunctorUtils.copy(predicates) : predicates;
        iClosures = clone ? FunctorUtils.copy(closures) : closures;
             iDefault = (Closure<? super E>) (defaultClosure == null ? NOPClosure.<E>nopClosure() :
defaultClosure);
}
```

It is obvious, from this, that SwitchedClosure also aggregates and inherits from Closure.

The next step is to find all dotted edges (i.e. RDs), as stated by p. 139 so as to take into consideration dynamic dependencies as well. C is a server class of X if “X associates with [C] or is an aggregated class of [C]” *(p. 139)*. A directed edge is drawn between C1 and C3 if there exists a class C2 that is both a server class of C1 and a parent class of C3 *(p. 139)*. In our case, we are able to see that, if we take C1 to be SwitchClosure, a dotted directed edge should be drawn towards CatchAndRethrowClosure, ChainedClosure, and SwitchClosure itself, since all of these

inherit from Closure. Hence, Closure is both a server class of SwitchClosure, and also a parent class of CatchAndRethrowClosure, ChainedClosure, and SwitchClosure. Similarly, for ChainedClosure, we have that Closure is both a server class of ChainedClosure, and also a parent class of CatchAndRethrowClosure, ChainedClosure, and SwitchClosure. Adding all of these factors together, we obtain the following "local" C-ORD:

Graph 1.1 - C-ORD for Classes of Closure Family



By inspection of the graph above, the derivation of the class firewalls becomes trivial, as we simply find, for each class X, "the set of classes that could be affected by changes to class" and which should be retested in case of modifications to X *(p. 137)*. For example, we notice that changes in Closure would imply re-testing of SwitchClosure, ChainedClosure and CatchAndRethrowClosure, since they all inherit from it. Following a similar approach for the rest of the classes, we obtain:

Table 1.2 - Kung et. Al Firewall Table for Classes of Closure Family

Class X	**CFW(X)**
Closure	SwitchClosure, ChainedClosure, CatchAndRethrowClosure
SwitchClosure	ChainedClosure
ChainedClosure	SwitchClosure
CatchAndRethrowClosure	ChainedClosure, SwitchClosure

Next, we need to build a dependency table. Let D1(X) be the "set of classes on which X depends statically," D2(X) the "set of classes on which X depends" statically and/or dynamically, and BD(X) the boolean function which specifies "whether or not X may dynamically depend on at least one class of D2(X) due to polymorphism" *(p. 139)*. We refer to a "solid path" as a path which traverses only non-dotted edges in the C-ORD above. We observe that all

classes have a directed "solid" path towards Closure, aside from Closure itself, which means that D1(SwitchClosure) = D1(ChainedClosure) = D1(CatchAndRethrowClosure) = {Closure} and D1(Closure) = Empty Set. Moving forward with our analysis, we consider all regular paths (i.e. including both dotted and non-dotted edges). We see that SwitchClosure and Chained Closure have paths to the same classes, implying that both their dynamic and static dependencies are the same. Hence: D2(SwitchClosure) = D2(ChainedClosure) = {Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}. CatchAndRethrowClosure didn't introduce any new dependencies due to polymorphism, therefore D2(CatchAndRethrowClosure) = D1(CatchAndRethrowClosure) = {Closure}. Clearly, D2(Closure) = D1(Closure) = Empty Set. Finally, we look for all classes X for which D1(X) is a strict subset of D2(X) to determine BD(X) the truth value of *(p. 139)*. In this case, BD(X) is only true for SwitchClosure and ChainedClosure, since D1(SwitchClosure) ⊂ D2(SwitchClosure) and D1(ChainedClosure) ⊂ D2(ChainedClosure). With all these factors taken into consideration, we get:

Table 1.3 - Dependency Analysis Table for Classes of Closure Family

Class X	**D1(X)**	**D2(X)**	**BD(X)**
Closure	ø	ø	0
SwitchClosure	{Closure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	1
ChainedClosure	{Closure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	1
CatchAndRethrowClosure	{Closure}	{Closure}	0

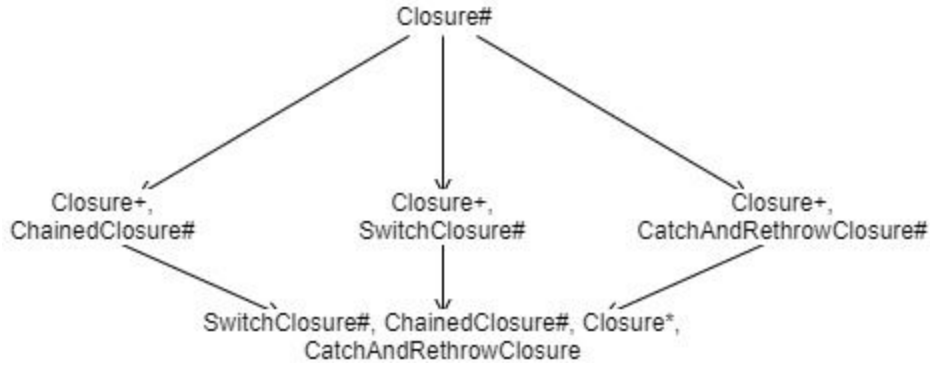
A detailed description of the particular strategy that needs to be adopted for deriving testing levels at this stage is stated on p. 140 of the paper. In short, we define T.goal as the "set of classes under test," T.need as "the set of classes that have to be involved in the test experiment", and T.type "the type of dependencies which are taken into account," which can be either static or dynamic (p. 140). We follow a systematic approach, wherein we, first, simply look at the table above and define, for all classes under test (i.e. all T .goals), corresponding T.needs by taking the union of class X and its set D1(X) for static dependencies. Then, to take into account dynamic dependencies, we group all dynamically dependent classes X (i.e. classes for which BD(X) = 1) by equivalent D2(X)s onto the same test level. For instance, we group both {SwitchClosure} and {ChainedClosure} because D2(SwitchClosure) = D2(ChainedClosure), and thus we obtain P = {{{SwitchClosure, ChainedClosure}, {Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}, Dyn}. Hence, by inspection, we define the following testing levels:

Table 1.4 - Testing Levels for Classes of Closure Family

T = (T.goal, T.need, T.Type)		
T.goal	**T.need**	**T.type**
{Closure}	{Closure}	Sta
{SwitchClosure}	{SwitchClosure, Closure}	Sta
{ChainedClosure}	{ChainedClosure, Closure}	Sta
{CatchAndRethrowClosure}	{CatchAndRethrowClosure, Closure}	Sta
{SwitchClosure, ChainedClosure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	Dyn

According to the rules defined in section 3.3 (*p. 140*), {{Closure}, {Closure}, Sta} should be preceding levels {{SwitchClosure}, {SwitchClosure, Closure}, Sta}, {{ChainedClosure}, {ChainedClosure, Closure}, Sta} and {{CatchAndRethrowClosure}, {CatchAndRethrowClosure, Closure}, Sta}, due to the fact that M.type = Sta = N1.type = N2.type = N3.type, and M.need ⊂ N1.need, M.need ⊂ N2.need, M.need ⊂ N3.need, whereby M = {{Closure}, {Closure}, Sta} and N1, N2, N3 = {{SwitchClosure}, {SwitchClosure, Closure}, Sta}, {{ChainedClosure}, {ChainedClosure, Closure}, Sta}, {{CatchAndRethrowClosure}, {CatchAndRethrowClosure, Closure}, Sta}. Similarly, because N1.type = Sta and P.type = Dyn, N2.type = Sta and P.type = Dyn, N3.type = Sta and P.type = Dyn, in addition to the fact that Nx.need ⊆ P.need, where x = 1, 2, 3, then N1, N2, N3 should be preceding P, where P = {{SwitchClosure, ChainedClosure}, {Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}, Dyn}. Hence, we get the following graphical representation by following the writing conventions specified in Y. Labiche et al.'s paper (*p. 141*):

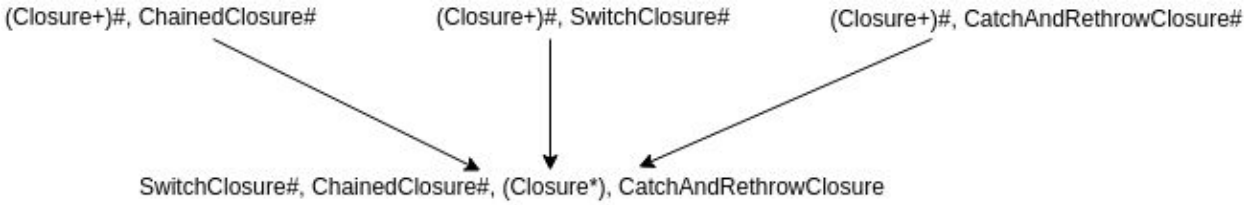
Graph 1.5 - Test Order Graph Using Writing Conventions for Classes of Closure Family



However, graph 1.4 does not account for the fact that the role of an abstract class at a given level might require its instantiation. In our case, this happens at Closure# on the first level, and at Closure+, CatchAndRethrowClosure# on

the second level. Hence, in the places affected, we have to delete the corresponding vertex and merge incoming and outgoing edges. Secondly, we also need to consider scenarios "where an abstract class in a testing level may be both a parent and a server class" *(p.142)*. In that case, we are instructed to instantiate the server relationship only with children of the abstract class (*p. 142*), and we indicate that by adding parentheses to the special characters "+" and "*" (*p. 142*). Thus, vertex for Closure# disappears, and the subsequent vertices become {**(**Closure+**)#**, ChainedClosure#}, {**(**Closure+**)#**, SwitchClosure#} and {**(**Closure+**)#**, CatchAndRethrowClosure#}. The final test order which takes into consideration abstract classes is, therefore:

Graph 1.6 - Test Order Graph Accounting for Abstract Classes for Classes of Closure Family



As you can notice, we kept the vertex for {Closure+, CatchAndRethrowClosure#} intact. This is due to the fact that, in the set of classes provided, no actual instantiation of CatchAndRethrowClosure is provided in the 20 original classes of the Apache Commons Collections tested in this deliverable. Therefore, we cannot postpone the test of the classes affected in the vertex, since CatchAndRethrowClosure cannot be tested at any of the subsequent testing levels. To solve this issue, we simply introduce a mock object of it.

Finally, Kung et al.'s order for the testing levels looks as such:

Table 1.7 - Kung et al.'s Testing Levels for Classes of Closure Family

Testing Level	**Class(es)**
1	Closure ChainedClosure SwitchClosure CatchAndRethrowClosure
2	SwitchClosure ChainedClosure

Bag Family

The “Classes of Bag Family” tag, which recurs throughout this section, refers to the set of dependencies between the following classes: Bag, SortedBag, and BagUtils.

Analysis of the source code begins our process. Starting with Bag, we immediately notice that Bag is an Interface, which has no dependencies on any of the 20 classes that we are analyzing. Continuing the process, it is seen that SortedBag is a subinterface of Bag. SortedBag also has no other dependencies on the 20 classes.

Upon inspection of BagUtils, it is determined that it is a basic Class which inherits none of the 20 classes we were assigned. Closer inspection reveals that BagUtils contains 2 static fields of the Bag type:

```
/**
 * An empty unmodifiable bag.
 */
@SuppressWarnings("rawtypes") // OK, empty bag is compatible with any type
public static final Bag EMPTY_BAG = UnmodifiableBag.unmodifiableBag(new HashBag<>());

/**
 * An empty unmodifiable sorted bag.
 */
@SuppressWarnings("rawtypes") // OK, empty bag is compatible with any type
public static final Bag EMPTY_SORTED_BAG =
        UnmodifiableSortedBag.unmodifiableSortedBag(new TreeBag<>());
```

As these fields are instantiated within the BagUtils class, it would seem that BagUtils is a composition of Bag. However, since these fields are static, any BagUtils instance points to the same 2 Bag objects, and deletion of any of these instances only deletes access to the Bag objects, not the objects themselves. This means that BagUtils associates with Bag, and the possibility of a composition can be nullified.

Continuing with the analysis of BagUtils, we see that some of the methods inside the class take objects of type Bag as argument (more association), and also some specifically take objects of type SortedBag as argument:

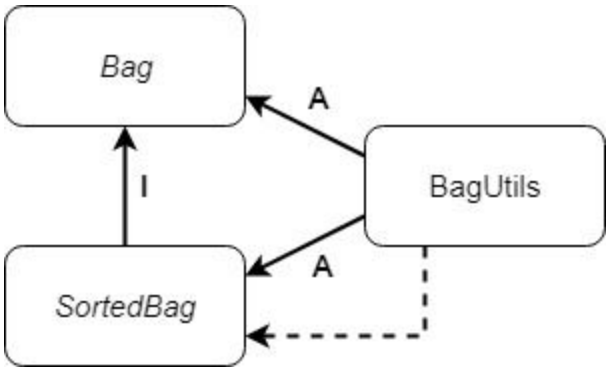
```
public static <E> SortedBag<E> synchronizedSortedBag(final SortedBag<E> bag) {
    return SynchronizedSortedBag.synchronizedSortedBag(bag);
}
```

This means that although SortedBag is a subinterface of Bag, and BagUtils already associates with Bag, BagUtils also directly associates with SortedBag. This is because a method of this type can take any object that implements SortedBag as argument, but it cannot take as argument an object that implements Bag and not SortedBag.

With all the static dependencies found for the three Classes of Bag Family, we move on to determining the dynamic dependencies. In this family there is only one; namely, BagUtils dynamically depends on SortedBag. This is because of the association between BagUtils and the parent interface Bag.

Adding all of these factors together, we obtain the following “local” C-ORD:

Graph 2.1 - C-ORD for Classes of Bag Family



Using the same process as defined in the Closure Family section, the class firewall table follows trivially:

Table 2.2 - Kung et. Al Firewall Table for Classes of Bag Family

Class X	**CFW(X)**
Bag	SortedBag, BagUtils
SortedBag	BagUtils
BagUtils	ø

Next, the dependency table must be built. It is observed that BagUtils and SortedBag have “solid” paths to Bag, while Bag has no “solid” path to any class. We also see that BagUtils has a “solid” path to SortedBag. This means D1(Bag) = empty set, and D1(SortedBag) = Bag, and D1(BagUtils) = Bag, SortedBag. Now, considering regular paths, we see that all paths remain the same, as the only dotted edge is parallel to one of the solid edges analyzed earlier. So, D2(X) = D1(X), where X is any of the 3 classes. For BD(X) we can quickly see that it is only true for BagUtils, as it is the only class with a dotted edge leaving it. With all these factors taken into consideration, we get:

Table 2.3 - Dependency Analysis Table for Classes of Bag Family

Class X	**D1(X)**	**D2(X)**	**BD(X)**
Bag	ø	ø	0
SortedBag	{Bag}	{Bag}	0
BagUtils	{Bag, SortedBag}	{Bag, SortedBag}	1

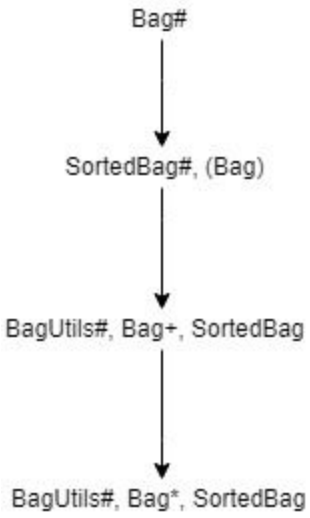
Following the process detailed in the Closure Family section, the testing levels table follows upon inspection of Table 2.3:

Table 2.4 - Testing Levels for Classes of Bag Family

T = (T.goal, T.need, T.Type)		
T.goal	**T.need**	**T.type**
{Bag}	{Bag}	Sta
{SortedBag}	{Bag, SortedBag}	Sta
{BagUtils}	{Bag, SortedBag, BagUtils}	Sta
{BagUtils}	{Bag, SortedBag, BagUtils}	Dyn

Once again following the procedures and logic presented in the Closure Family section, and using the writing conventions specified in Y. Labiche et al.'s paper (*p. 141*), we can define an initial test order graph as illustrated in Graph 2.5:

Graph 2.5 - Test Order Graph Using Writing Conventions for Classes of Bag Family



As interfaces cannot be instantiated, much like abstract classes, the two are treated in the same manner. We see that the Bag and SortedBag interfaces are required to be instantiated in all levels. This means that the first and second levels are merged, allowing the Bag test to be performed in parallel with the SortedBag test. However, SortedBag still must be instantiated, so this new testing level is dropped to the next feasible level (this is a merging of levels 1, 2, and 3). We cannot merge the static and dynamic tests for Bag, and so we are left with two testing levels, defined in Graph 2.6:

Graph 2.6 - Test Order Graph Accounting for Interfaces for Classes of Bag Family

These levels still have issues, however. Although in both cases the Bag class is only used through its children, its only child that we are supposed to take into account for the purposes of this project is SortedBag. SortedBag is itself an interface. This means that we have 2 remaining instantiation problems, both of which occur in both levels: Bag must be instantiated through its child SortedBag (an interface), and SortedBag must itself be instantiated. This problem is easily solved by using a Mock object that implements the SortedBag interface, as this one mock object can be used for the Bag and SortedBag tests.

Finally, Kung et al.'s order for the testing levels is shown in Table 2.7.

Table 2.7 - Kung et al.'s Testing Levels for Classes of Bag Family

Testing Level	**Class(es)**
1	Bag, SortedBag, BagUtils
2	BagUtils

Command Family

The “Classes of Command Family” tag which recurs throughout this section refers to the set of dependencies between the following classes: EditCommand, DeleteCommand, InsertCommand, KeepCommand.

• EditCommand <Abstract>

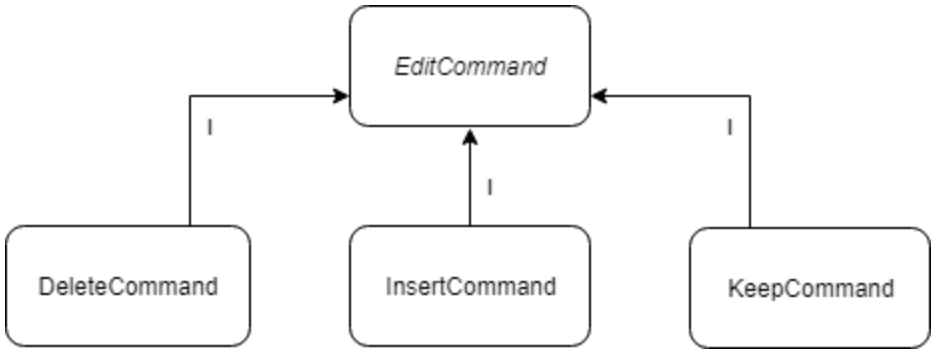
• KeepCommand
    extends EditCommand
        inherits getObject()
• InsertCommand
    extends EditCommand
        inherits getObject()
• DeleteCommand
    extends EditCommand
        inherits getObject()

As shown above, the EditCommand class is parent to all other classes in the classes of command family. KeepCommand, InsertCommand and DeleteCommand are independent of one another. The result is a very simple dependency structure with all three classes inheriting from their common parent. Each class mentionned above also has an association with class CommandVisitor through the method accept(CommandVisitor<T> visitor) as well as a composition relationship with Generic Type object through their inherited constructor. However, these classes are outside of the scope of this assignment and will therefore not be taken into account.

```
public abstract class EditCommand<T> {
…
     private final T object;
…
protected EditCommand(final T object) {
   this.object = object;
}
…
protected T getObject() {
  return object;
}
…
public abstract void accept(CommandVisitor<T> visitor);
```

Graph 3.1 below consists only of inheritance relationships hence the absence of any dotted lines representing polyphormism.

Graph 3.1 - C-ORD for Classes of Command Family



From the graph above, and following the methodology described in the previous sections, the tables below trivially follow:

Table 3.2 - Kung et. Al Firewall Table for Classes of Command Family

Class X	**CFW(X)**
EditCommand	DeleteCommand InsertCommand KeepCommand
DeleteCommand	
InsertCommand	
KeepCommand	

Table 3.3 - Dependency Analysis Table for Classes of Command Family

Class X	**D1(X)**	**D2(X)**	**Bd(X)**
EditCommand	ø	ø	0
DeleteCommand	{EditCommand}	{EditCommand}	0
InsertCommand	{EditCommand}	{EditCommand}	0
KeepCommand	{EditCommand}	{EditCommand}	0

Table 3.4 - Testing Levels for Classes of Command Family

T = (T.goal, T.need, T.type)		
T.goal	T.need	T.type
{EditCommand}	{EditCommand}	Sta
{DeleteCommand}	{DeleteCommand, EditCommand}	Sta
{InsertCommand}	{InsertCommand, EditCommand}	Sta
{KeepCommand}	{KeepCommand, EditCommand}	Sta

All testing levels are static. As a result, each test for a specific class only needs itself and its parent class. However, because its parent class is abstract, the instantiation of the class under test will satisfy both those needs.

Graph 3.5 - Test Order Graph Using Writing Conventions for Classes of Command Family

Graph 3.6 - Test Order Graph Accounting for Abstract Classes for Classes of Command Family

{DeleteCommand#, (EditCommand+)} {InsertCommand#, (EditCommand+)} {KeepCommand#, (EditCommand+)}

EditCommand is an abstract parent class for DeleteCommand, InsertCommand, KeepCommand as indicated by the parenthesis accompanied by the "+" sign. The classes under test in each bracket is indicated by the "#" sign.

EditCommand's initial level was removed between the C-ORD and the TOG because of its abstract nature. Any test to DeleteCommand, InsertCommand or KeepCommand will test EditCommand through instantiation of an abstract parent class. Therefore, DeleteCommand, InsertCommand, KeepCommand and EditCommand can be tested on the same level as shown in graph 3.6.

Table 3.7 - Kung et al.'s Testing Levels for Classes of Command Family

Testing Level	Class(es)
1	DeleteCommand, InsertCommand, KeepCommand

MapIterator Family

The Classes of MapIterator family refer to the following set of classes: OrderedMapIterator, AbstractMapIteratorDecorator, UnmodifiableMapIterator, MapIterator, EmptyOrderedMapIterator and EmptyMapIterator.

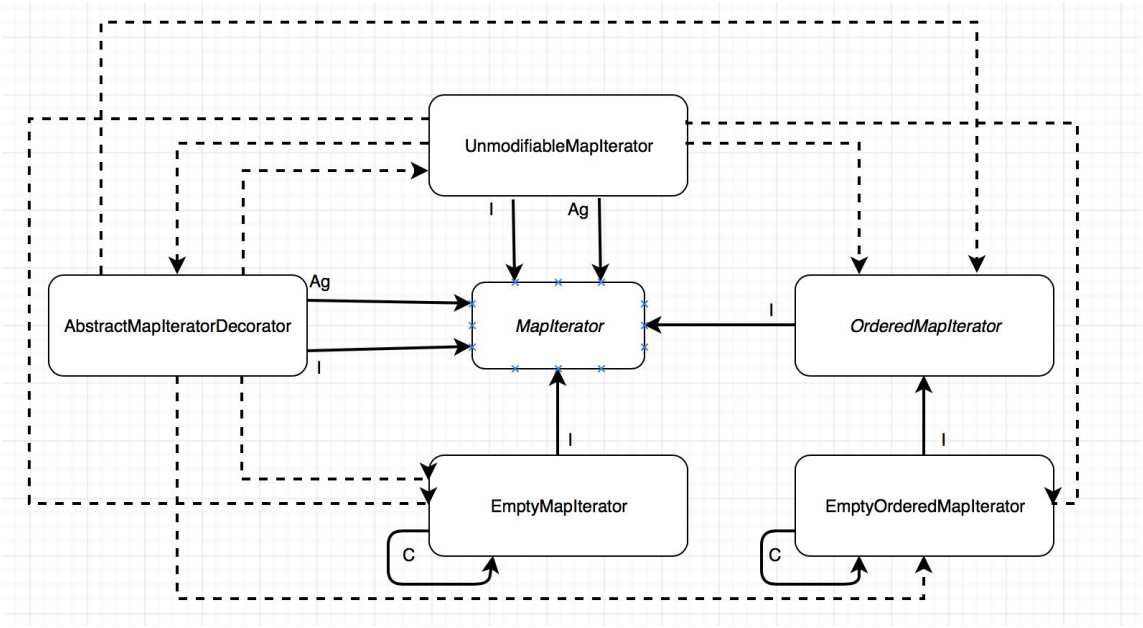
After further investigation into each one of these classes we've noticed that MapIterator and OrderedMapIterator are interfaces whereas the rest are classes. Moreover, the MapIterator is a parent class to AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, and EmptyMapIterator. Similarly, OrderedMapIterator is a parent of EmptyOrderedMapIterator.

Additionally, we notice that there's a dependency injection of MapIterator inside of AbstractMapIteratorDecorator and UnmodifiableMapIterator:

```
private final MapIterator<K, V> iterator;
...
public AbstractMapIteratorDecorator(final MapIterator<K, V> iterator) {
     super();
     if (iterator == null) {
         throw new NullPointerException("MapIterator must not be null");
     }
     this.iterator = iterator;
}
```

```
private final MapIterator<? extends K, ? extends V> iterator;
...
private  UnmodifiableMapIterator(final  MapIterator<?  extends  K,  ?  extends  V>
iterator) {
    super();
    this.iterator = iterator;
}
```

Graph 4.1 - C-ORD for Classes of MapIterator Family



By inspecting the previous graph, we were able to derive the firewall by pursuing the following the logic stated in the previous Closure Family section:

Table 4.2 - Kung et. Al Firewall Table for Classes of MapIterator Family

Class X	**CFW(X)**
MapIterator	EmptyMapIterator, AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator
EmptyMapIterator	ø
AbstractMapIteratorDecorator	ø
UnmodifiableMapIterator	ø
OrderedMapIterator	EmptyOrderedMapIterator
EmptyOrderedMapIterator	ø

Next, we need to build a dependency table. Let D1(X) be the "set of classes on which X depends statically," D2(X) the "set of classes on which X depends" statically and/or dynamically, and BD(X) the boolean function which specifies "whether or not X may dynamically depend on at least one class of D2(X) due to polymorphism" *(p. 139)*. AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator and EmptyMapIterator are all subclasses of MapIterator. Similarly, EmptyOrderedMapIterator is both a subclass of OrderedMapIterator and MapIterator. Moreover, Since AbstractMapIteratorDecorator and UnmodifiableMapIterator are both aggregated classes of MapIterator and MapIterator is a parent class of EmptyMapIterator, AbstractMapIteratorDecorator,

UnmodifiableMapIterator, OrderedMapIterator and EmptyOrderedMapIterator, we get the following static and dynamic dependencies

Table 4.3 - Dependency Analysis Table for Classes of MapIterator Family

Class X	D1(X)	D2(X)	BD(X)
MapIterator	ø	ø	0
EmptyMapIterator	{MapIterator}	{MapIterator}	0
AbstractMapIteratorDecorator	{MapIterator}	{MapIterator, EmptyMapIterator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator}	1
UnmodifiableMapIterator	{MapIterator}	{MapIterator, EmptyMapIterator, AbstractMapIteratorDecorator, OrderedMapIterator, EmptyOrderedMapIterator}	1
OrderedMapIterator	{MapIterator}	{MapIterator}	0
EmptyOrderedMapIterator	{MapIterator, OrderedMapIterator}	{MapIterator, OrderedMapIterator}	0

A detailed description of the particular strategy that needs to be adopted for deriving testing levels at this stage is stated on p. 140 of the paper. In short, we define T.goal as the "set of classes under test," T.need as "the set of classes that have to be involved in the test experiment", and T.type "the type of dependencies which are taken into account," which can be either static or dynamic (p. 140). We follow a systematic approach, wherein we, first, simply look at the table above and define, for all classes under test (i.e. all T .goals), corresponding T.needs by taking the union of class X and its set D1(X) for static dependencies. Then, to take into account dynamic dependencies, we group all dynamically dependent classes X (i.e. classes for which BD(X) = 1) by equivalent D2(X)s onto the same test level.

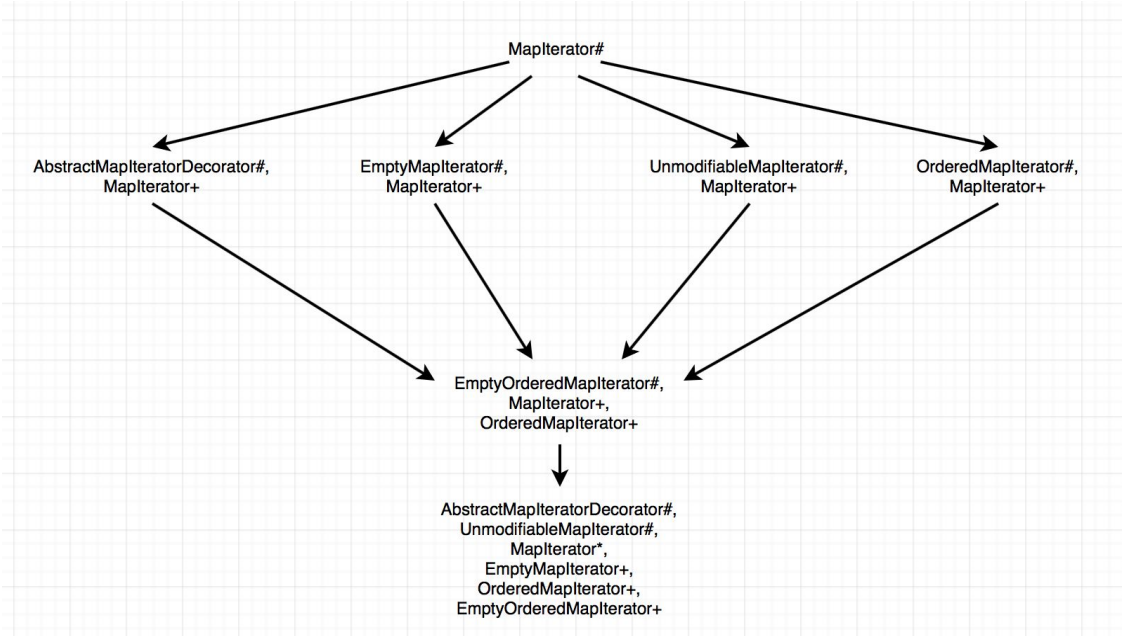
According to the rules defined in section 3.3 (*p. 140*), {{MapIterator}, {MapIterator}, Sta} should be preceding levels {{EmptyMapIterator}, {EmptyMapIterator, MapIterator}, Sta}, {{AbstractMapIteratorDecorator}, {AbstractMapIteratorDecorator, MapIterator}, Sta}, {{UnmodifiableMapIterator}, {UnmodifiableMapIterator, MapIterator}, Sta} and {{OrderedMapIterator}, {OrderedMapIterator, MapIterator}, Sta}, due to the fact that L.type = Sta = M1.type = M2.type = M3.type = M4.type, and L.need ⊂ M1.need, L.need ⊂ M2.need, L.need ⊂ M3.need, L.need ⊂ M4.need whereby L = {{MapIterator}, {MapIterator}, Sta} and M1, M2, M3, M4 = {{EmptyMapIterator}, {EmptyMapIterator, MapIterator}, Sta}, {{AbstractMapIteratorDecorator}, {AbstractMapIteratorDecorator, MapIterator}, Sta}, {{UnmodifiableMapIterator}, {UnmodifiableMapIterator, MapIterator}, Sta} and {{OrderedMapIterator}, {OrderedMapIterator, MapIterator}, Sta}. Similarly, M1, M2, M3, and M4 should be preceding N = {{EmptyOrderedMapIterator}, {EmptyOrderedMapIterator, MapIterator}, Sta}, due to the fact that M1.type = M2.type = M3.type = M4.type = Sta = N.type, and M1.need ⊂ N.need, M2.need ⊂ N.need, M3.need ⊂ N.need, M4.need ⊂ N.need. Finally, because P.type = Dyn and and L.type = Mx.type = Sta, in addition to the fact that L.need ⊆ P.need and Mx.need ⊆ P.need, where x = 1, 2, 3, 4 and P = {{AbstractMapIteratorDecorator, UnmodifiableMapIterator}, {MapIterator, EmptyMapIterator,

AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator}, Dyn}, then L, M1, M2, M3, and M4 should be preceeding P. Hence, we get the following graphical representation by following the writing conventions specified in Y. Labiche et al.'s paper (*p. 141*):

Table 4.4 - Testing Levels for Classes of MapIterator Family

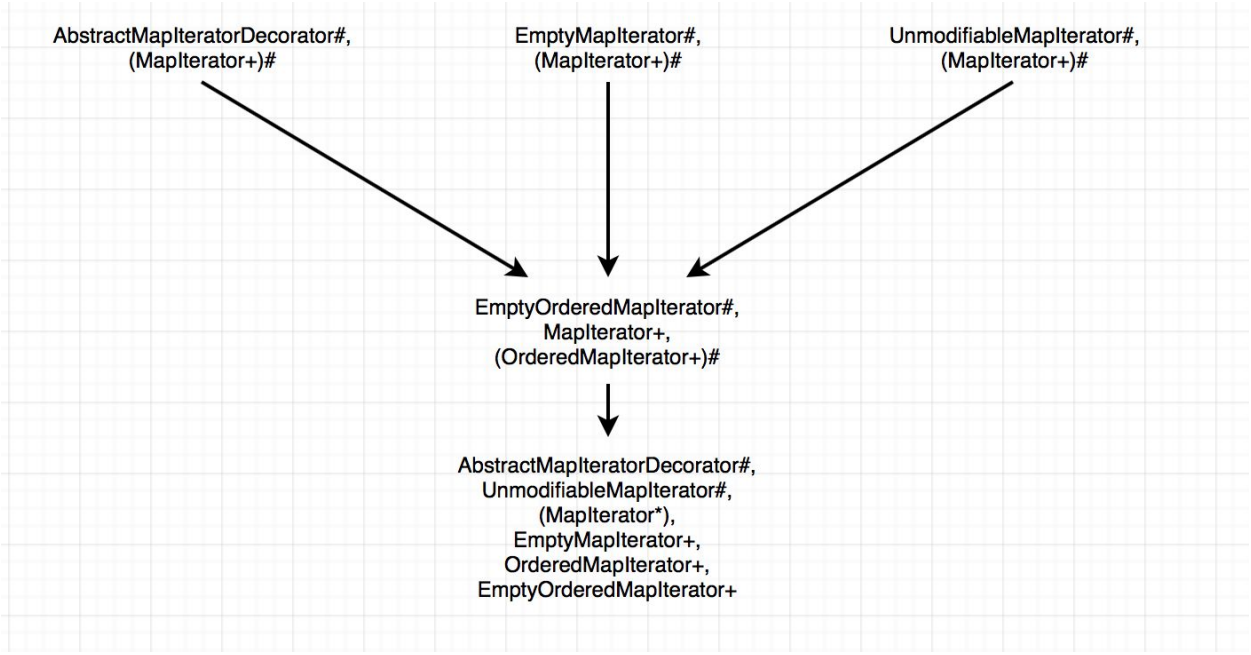
T = (T.goal, T.need, T.type)		
T.goal	**T.need**	**T.type**
MapIterator	{MapIterator}	Sta
EmptyMapIterator	{EmptyMapIterator, MapIterator}	Sta
AbstractMapIteratorDecorator	{AbstractMapIteratorDecorator, MapIterator}	Sta
UnmodifiableMapIterator	{UnmodifiableMapIterator, MapIterator}	Sta
OrderedMapIterator	{OrderedMapIterator, MapIterator}	Sta
EmptyOrderedMapIterator	{EmptyOrderedMapIterator, MapIterator, OrderedMapIterator}	Sta
AbstractMapIteratorDecorator, UnmodifiableMaptIterator	{MapIterator, EmptyMapIterator, AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator}	Dyn

Graph 4.5 - Test Order Graph Using Writing Conventions for Classes of MapIterator Family



However, graph 2 does not account for the fact that the role of an abstract class at a given level might require its instantiation. In our case, this happens at MapIterator# on the first level, and at OrderedMapIterator#, MapIterator+ on the second level. Hence, in the places affected, we have to delete the corresponding vertex and merge incoming and outgoing edges. Secondly, we also need to consider scenarios "where an abstract class in a testing level may be both a parent and a server class" *(p.142)*. In that case, we are instructed to instantiate the server relationship only with children of the abstract class (*p. 142*), and we indicate that by adding parentheses to the special characters "+" and "*" (*p. 142*). Thus, vertex for MapIterator# disappears, and the subsequent vertices become {AbstractMapIteratorDecorator#, **(**MapIterator+**)#**}, {EmptyMapIterator#, **(**MapIterator+**)#**}, {UnmodifiableMapIterator, **(**MapIterator+**)#**}, and {OrderedMapIterator#, (MapIterator+)#}. The final test order which takes into consideration abstract classes is, therefore:

Graph 4.6 - Test Order Graph Accounting for Abstract Classes for Classes of MapIterator Family



MapIterator and OrderedMapIterator are both interfaces, and, similarly to abstract classes, cannot be instantiated. We can see that MapIterator must be instantiated in all levels. Therefore, The first and second level are merged in order to perform the MapIterator test and OrderedMapIterator test in parallel. However, OrderedMapIterator still has to be instantiated on the second level and is thus dropped to the next feasible level. Finally, since MapIterator can be tested through AbstractMapIteratorDecorator, EmptyMapIterator, and UnmodifiableMapIterator we do not need any mocks or stubs. Similarly, OrderedMapIterator can be tested through EmptyOrderedMapIterator, and, thus, no mocks or stubs are needed for it.

Finally, Kung et al.'s order for the testing levels looks as such:

Table 4.7 - Kung et al.'s Testing Levels for Classes of MapIterator Family

Testing Level	**Class(es)**
1	MapIterator AbstractMapIteratorDecorator EmptyMapIterator UnmodifiableIterator
2	EmptyOrderedMapIterator OrderedMapIterator
3	AbstractMapIteratorDecorator UnmodifiableMapIterator

Trie Family

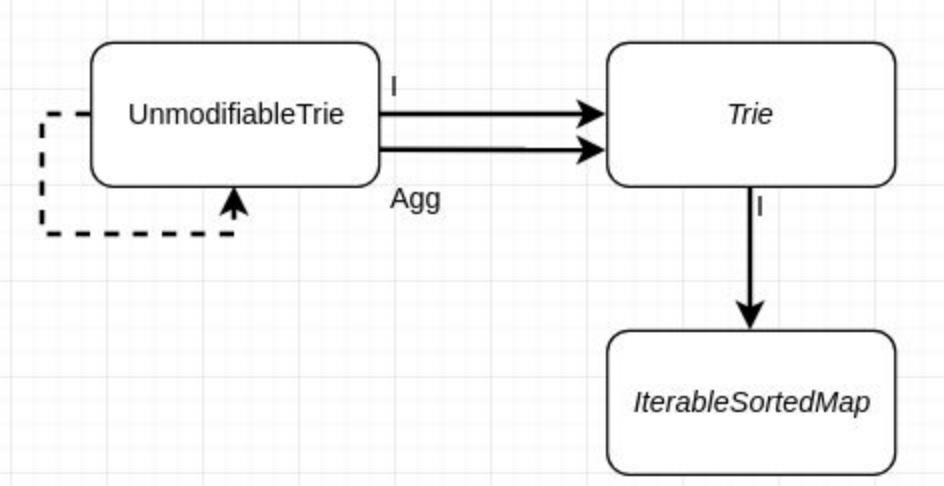
The Classes of Trie Family refer to the following set of classes: Trie, UnmodifiableTrie and IterableSortedMap. After inspection we noticed that Trie and IterableSortedMap are interfaces whereas UnmodifiableTrie is a class. UnmodifiableTrie implements Trie, which in turn implements IterableSortedMap.

Similarly, we notice that there's a dependency injection of Trie inside of UnmodifiableTrie:

```
private final Trie<K, V> delegate;
...
 public UnmodifiableTrie(final Trie<K, ? extends V> trie) {
        if (trie == null) {
            throw new NullPointerException("Trie must not be null");
        }
        @SuppressWarnings("unchecked") // safe to upcast
        final Trie<K, V> tmpTrie = (Trie<K, V>) trie;
        this.delegate = tmpTrie;
}
```

This implies that there is an aggregation relationship between the two classes. We therefore obtain the following ORD:

Graph 5.1 - C-ORD for Classes of Trie Family



As we can see, there is only a single polymorphic dependency, because, since UnmodifiableTrie implements Trie, it can also be passed as a parameter to its own constructor. By inspecting the previous graph, we were able to derive the firewall by pursuing the following the logic stated in the previous Closure Family section:

Table 5.2 - Kung et. Al Firewall Table for Classes of Trie Family

Class X	CFW(X)
IterableSortedMap	UnmodifiableTrie, Trie
UnmodifiableTrie	ø
Trie	UnmodifiableTrie

Next, we need to build a dependency table by following the conventions established in previous sections for D1(X), D2(X) and BD(X). UnmodifiableTrie is a subclass of both Trie and IterableSortedMap. Similarly, Trie is a child of IterableSortedMap. Since there are no dynamic dependencies, we observe that for all X != UnmodifiableTrie in the family of classes, D1(X) and D2(X) are equivalent sets. In the case of UnmodifiableTrie, we can see that it has a dynamic dependency of itself. Thus, we get the following dependency table:

Table 5.3 - Dependency Analysis Table for Classes of Trie Family

Class X	D1(X)	D2(X)	BD(X)
IterableSortedMap	ø	ø	0
Trie	{IterableSortedMap}	{IterableSortedMap}	0
UnmodifiableTrie	{IterableSortedMap, Trie}	{IterableSortedMap, Trie, UnmodifiableTrie}	1

The next part requires developing the testing levels. To do so we need to define the T.goals as well as the T.needs , and the T.types for each class under test and we follow the same systematic approach that we did in the Closure Family section. This time, we get the particular case where a testing level's type is both dynamic and static, due to the fact that the dynamic dependency for UnmodifiableTrie is introduced by itself, and thus there is no need to retest the class on another level to ensure its correctness, since it can be done simultaneously. Thus:

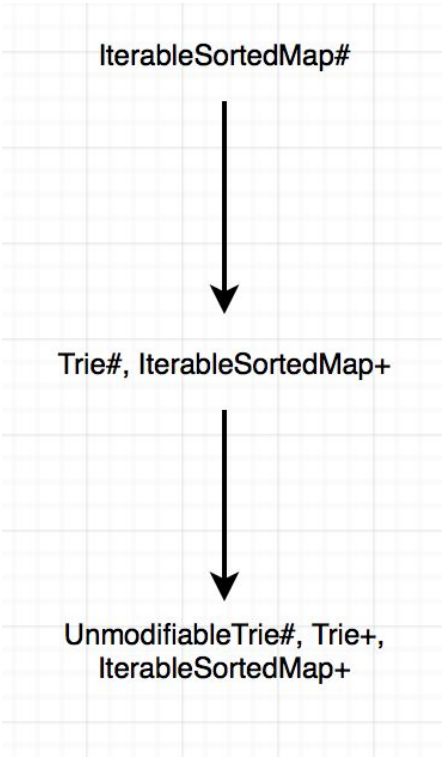
Table 5.4 - Testing Levels for Classes of Trie Family

T = (T.goal, T.need, T.type)		
T.goal	**T.need**	**T.type**
IterableSortedMap	{IterableSortedMap}	Sta
Trie	{Trie, IterableSortedMap}	Sta
UnmodifiableTrie	{Trie, UnmodifiableTrie, IterableSortedMap}	Sta / Dyn

According to the rules defined in section 3.3 (*p. 140*), {{IterableSortedMap}, {IterableSortedMap}, Sta} should be preceding levels {{Trie}, {Trie, IterableSortedMao}, Sta}, and {{UnmodifiableTrie}, {UnmodifiableTrie,

IterableSortedMap}, Sta}, due to the fact that L.type = Sta = M.type, and L.need ⊂ M.need whereby L = {{IterableSortedMap}, {IterableSortedMap}, Sta} and M= {{Trie}, {Trie, IterableSortedMao}, Sta}. Similarly, M.type = Sta = N.type, and M.need ⊂ N.need whereby N = {{UnmodifiableTrie}, {UnmodifiableTrie, Trie, IterableSortedMap}, Sta / Dyn}. Hence, we get the following graphical representation by following the writing conventions specified in Y. Labiche et al.'s paper (*p. 141*):

Graph 5.5 - Test Order Graph Using Writing Conventions for Classes of MapIterator Family



However, from the above, we quickly realize that our first and second testing levels are infeasible since we cannot instantiate interfaces at their respective levels. Therefore, we must merge the infeasible levels with subsequent ones, until IterableSortedMap and Trie become testable, and this is done by merging both the IteratableSortMap# and Trie#, IteratableSortedMap+ levels into one, final level called (UnmodifiableTrie)#, (Trie+)#, (InteratableSortedMap+)#:

Graph 5.6 - Test Order Graph Accounting for Abstract Classes for Classes of MapIterator Family

(UnmodifiableTrie)#, (Trie+)#, (InteratableSortedMap+)#

Since both Trie and InteratableSortedMap can be tested through UnmodifiableTrie (due to the fact that it is a subclass of both), no mocks or stubs are needed for it.

Finally, Kung et al.'s order for the testing levels looks as such:

Table 5.7 - Kung et al.'s Testing Levels for Classes of Trie Family

Testing Level	Class(es)
1	IterableSortedMap, Trie, UnmodifiableTrie

Combined Tables for All Classes Examined

Firewalls

Table 6.1 - Global Kung et. Al Firewall Table for All Classes

Class X	**CFW(X)**
MapIterator	EmptyMapIterator, AbstarctMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator
EmptyMapIterator	ø
AbstarctMapIteratorDecorator	ø
UnmodifiableMapIterator	ø
OrderedMapIterator	EmptyOrderedMapIterator
EmptyOrderedMapIterator	ø
IterableSortedMap	UnmodifiableTrie, Trie
UnmodifiableTrie	ø
Trie	UnmodifiableTrie
EditCommand	DeleteCommand, KeepCommand, InsertCommand
DeleteCommand	ø
InsertCommand	ø
KeepCommand	ø
Bag	SortedBag, BagUtils
SortedBag	BagUtils
BagUtils	ø
Closure	SwitchClosure, ChainedClosure, CatchAndRethrowClosure
SwitchClosure	ChainedClosure
ChainedClosure	SwitchClosure
CatchAndRethrowClosure	ChainedClosure, SwitchClosure

Dependency Analysis

Table 6.2 - Global Dependency Analysis Table for All Classes

Class X	D1(X)	D2(X)	BD(X)
MapIterator	ø	ø	0
EmptyMapIterator	{MapIterator}	{MapIterator}	0
AbstractMapIteratorDecorator	{MapIterator}	{MapIterator, EmptyMapIterator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator}	1
UnmodifiableMapIterator	{MapIterator}	{MapIterator, EmptyMapIterator, AbstractMapIteratorDecorator, OrderedMapIterator, EmptyOrderedMapIterator}	1
OrderedMapIterator	{MapIterator}	{MapIterator}	0
EmptyOrderedMapIterator	{MapIterator, OrderedMapIterator}	{MapIterator, OrderedMapIterator}	0
IterableSortedMap	ø	ø	0
Trie	{IterableSortedMap}	{IterableSortedMap}	0
UnmodifiableTrie	{IterableSortedMap, Trie}	{IterableSortedMap, Trie, UnmodifiableTrie}	1
EditCommand	ø	ø	0
DeleteCommand	EditCommand	EditCommand	0
InsertCommand	EditCommand	EditCommand	0
KeepCommand	EditCommand	EditCommand	0
Bag	ø	ø	0
SortedBag	{Bag}	{Bag}	0
BagUtils	{Bag, SortedBag}	{Bag, SortedBag}	1
SwitchClosure	{Closure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	1

ChainedClosure	{Closure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	1
CatchAndRethrowClosure	{Closure}	{Closure}	0

Testing Levels:

Table 6.3 - Global Table for Labiche et Al. Testing Levels

T = (T.goal, T.need, T.type)		
T.goal	**T.need**	**T.type**
{MapIterator}	{MapIterator}	Sta
{EmptyMapIterator}	{EmptyMapIterator, MapIterator}	Sta
{AbstractMapIteratorDecorator}	{AbstractMapIteratorDecorator, MapIterator}	Sta
{UnmodifiableMapIterator}	{UnmodifiableMapIterator, MapIterator}	Sta
{OrderedMapIterator}	{OrderedMapIterator, MapIterator}	Sta
{EmptyOrderedMapIterator}	{EmptyOrderedMapIterator, MapIterator, OrderedMapIterator}	Sta
{AbstractMapIteratorDecorator, UnmodifiableMaptIterator}	{MapIterator, EmptyMapIterator, AbstractMapIteratorDecorator, UnmodifiableMapIterator, OrderedMapIterator, EmptyOrderedMapIterator}	Dyn
{IterableSortedMap}	{IterableSortedMap}	Sta
{Trie}	{Trie, IterableSortedMap}	Sta
{UnmodifiableTrie}	{Trie, UnmodifiableTrie, IterableSortedMap}	Sta / Dyn
{EditCommand}	{EditCommand}	Sta
{DeleteCommand}	{DeleteCommand, EditCommand}	Sta
{InsertCommand}	{InsertCommand, EditCommand}	Sta
{KeepCommand}	{DeleteCommand, EditCommand}	Sta

{Bag}	{Bag}	Sta
{SortedBag}	{Bag, SortedBag}	Sta
{BagUtils}	{Bag, SortedBag, BagUtils}	Sta
{BagUtils}	{Bag, SortedBag, BagUtils}	Dyn
{Closure}	{Closure}	Sta
{SwitchClosure}	{SwitchClosure, Closure}	Sta
{ChainedClosure}	{ChainedClosure, Closure}	Sta
{CatchAndRethrowClosure}	{CatchAndRethrowClosure, Closure}	Sta
{SwitchClosure, ChainedClosure}	{Closure, ChainedClosure, SwitchClosure, CatchAndRethrowClosure}	Dyn

Final Order of Testing Levels:

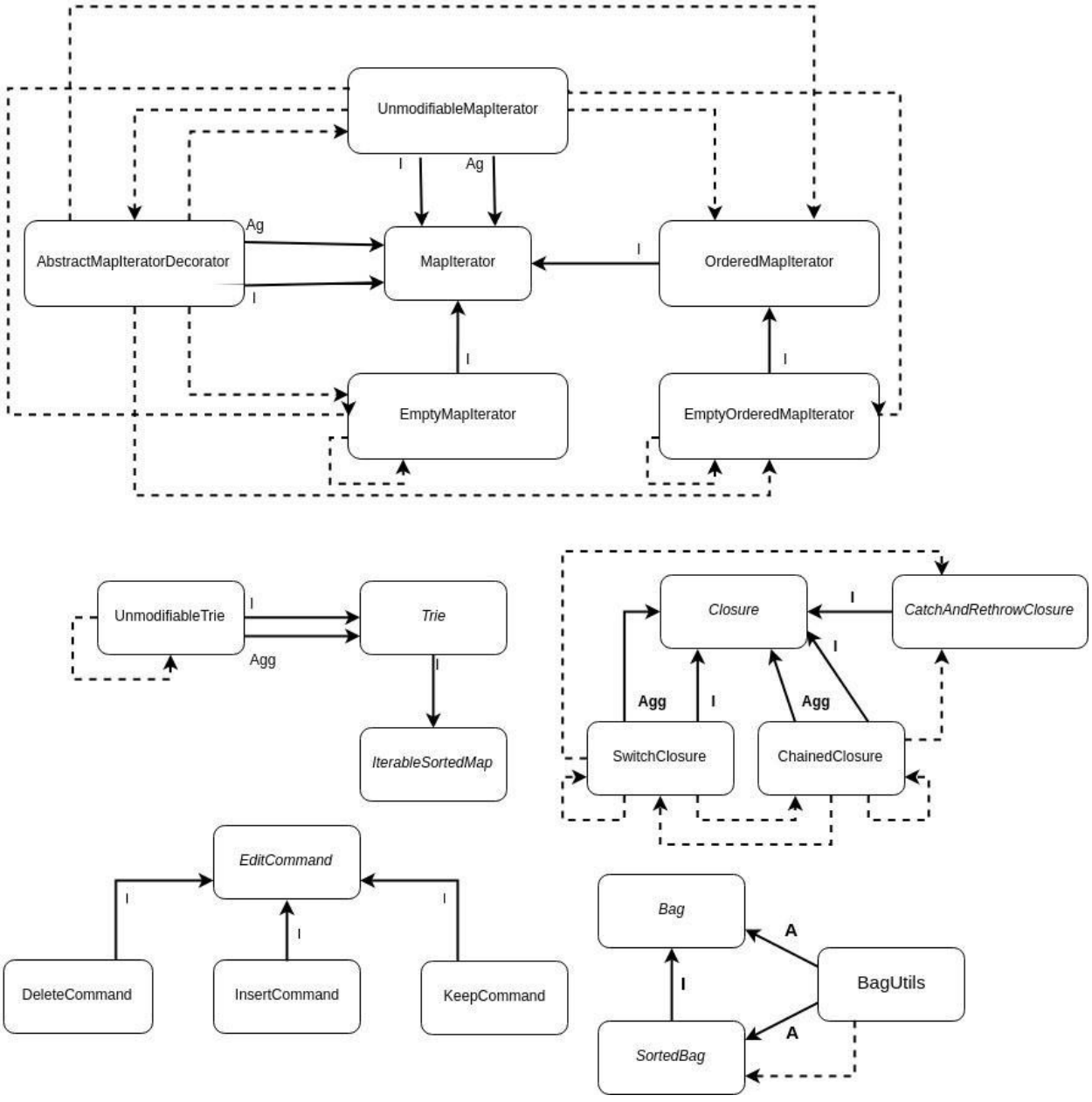
Table 6.4 - Kung et al.'s Testing Levels for All Classes

Testing Level	**Class(es)**
1	MapIterator Closure ChainedClosure SwitchClosure CatchAndRethrowClosure Bag SortedBag BagUtils IterableSortedMap Trie UnmodifiableTrie EditCommand DeleteCommand InsertCommand KeepCommand
2	AbstractMapIteratorDecorator EmptyMapIterator UnmodifiableIterator OrderedMapIterator BagUtils SwitchClosure ChainedClosure

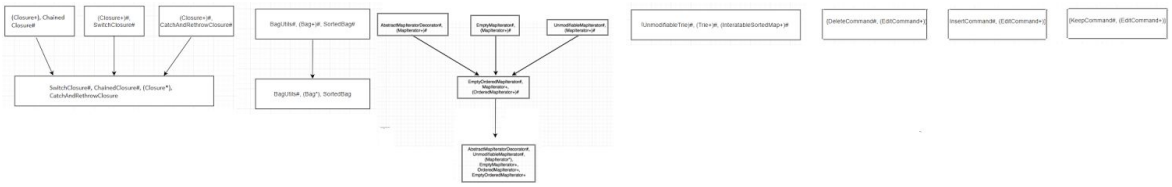
3	EmptyOrderedMapIterator

Final Graphs

Graph 6.5 - C-ORD for All Classes



Graph 6.6 - Testing Order for All Classes



PLEASE NOTE THAT SINCE THIS IMAGE IS TOO BIG, YOU'RE UNLIKELY TO BE ABLE TO SEE IT. WE HAVE SUBMITTED A STANDALONE FILE ON MYCOURSES, OR YOU CAN SIMPLY REFER TO EACH INDIVIDUAL TEST ORDER GRAPH IN ITS CORRESPONDING SECTION.

Mocks, Drivers, and Stubs

The Kung et al. strategy leads to a "bottom-up" integration test order. This allows the lowest level modules (which need no stubs) to be tested first, and then used in the higher level tests, completely removing the need for stubs. As such, our integration test order requires no stubs.

There were 2 cases where the designed tests were infeasible due to required instantiation of an abstract class and an interface. These occurred in the Closure family, and the Bag family. In both cases, the issue is solvable using mock objects. A mock object can be made that inherits the uninstantiable module's functionality, which enables the instantiation of a child class. This allows us to test the parent's functionality without actually instantiating it. Our integration tests would require 2 mock objects.

Although the "bottom-up" approach requires no stubs, it does require drivers. The required drivers could vary in complexity and number. A single driver could be made for each testing level, however, this kind of driver would be complex and cumbersome (needing to drive a large number of modules at one time. Instead of this, we have decided to use drivers for each test at each test level (following the test orders taking abstract classes into account). For example, in the Closure family's first level there would be a driver for the {(Closure+)#, ChainedClosure#} test, another for the {(Closure+)#, SwitchClosure#} test, and one more for the {(Closure+)#, CatchAndRethrowClosure#} test. These drivers would be individualized to each test case (allowing parallel development), and would also be simple enough to limit technical issues caused by attempting to control too much at once. Following this driver strategy, the total driver and mock object requirements are as follows:

Table 7.1 - Listing of Mocks and Drivers

Family	Number of Mocks	Number of Drivers
Closure	1	4
Bag	1	2
MapIterator	0	5

Trie	0	1
Command	0	3

Therefore, we require a total of 0 stubs, 2 mocks, and 15 drivers to complete our integration testing.