**NOTE IMPORTANTI:**

* La classe TestRunner permette, mediante l’uso delle Reflections, di eseguire tutte le suite di test insieme oppure di eseguire solo quelle desiderate indicando da riga di comando i nomi delle classi di test che vogliono essere eseguite.  
  Dalla cartella principale del progetto (dopo aver compilato e inserito Junit e hamcrest nel CLASSPATH):
  + **Java myTest\TestRunner.java***:* esegue tutti I test definiti riportando l’opportuno report dei risultati dell’esecuzione dei test
  + **Java myTest\TestRunner.java myTest.suite1 myTest.suite2 myTest.suite3:**

Permette di eseguire le suite “suite1”, “suite2”, “suite3” definite all’interno del package “myTest”. Il numero di suite eseguibili non ha un limite.

Un nome di suite errato lancia eccezione e fa terminare il programma.

* Tutta la documentazione delle classi, delle interfacce, dei casi di test e delle suite di test è  
  stata eseguita interamente con Javadoc.  
  Sono stati definiti tag custom per i vari campi inseriti all’interno della tabella fornita nella consegna dell’homework
* I risultati dei test sono stati forniti in formato HTML (suddivisi per suite) all’interno della cartella “Test Suite Execution Records”.  
  E’ inoltre possibile accedere a tali HTML anche direttamente dalla documentazione javadoc dei test aprendo il file .html javadoc della suite desiderata e cliccando sul link inserito sotto la sezione “execution records”.

La struttura delle suite di test è stata realizzata nel seguente modo:

Si è deciso di suddividere in questo modo le varie suite in modo da fornire la possibilità di testare in modo isolato le varie funzionalità.

Sono state create delle suite “TestSuiteList” e “TestSuiteMap” con il solo scopo di raggruppare le suite relative al ListAdapter o al MapAdapter e poterle eseguire in modo agevole.

Si è ottenuto questo risultato utilizzando i comandi @RunWith e @Suite.SuiteClasses definite dal framework JUnit che permette di eseguire più classi di test invocandone soltanto una vuota usata solo come holder per le annotazioni appena citate.

Informazioni sui pattern (GOF e non) utilizzzati:

**Adapter Pattern:** Questo structural pattern ha permesso di utilizzare l’interfaccia di una classe già esistente (adaptee) per realizzare una certa interfaccia (target). Tale pattern risulta essere utile quando una classe non può essere utilizzata solo perchè la sua interfaccia non è conforme all’interfaccia desideratadal cliente.

In particolare, si è utilizzato un oggetto Vector per realizzare l’iterfaccia HList e un oggetto di tipo Hashtable per realizzare l’interfaccia HMap.

In entrambi i casi si è optato per l’utilizzo di un Object Adapter al posto del Class Adapter. Tale scelta implementativa è principalmente legata al fatto di voler nascondere l’interfaccia dell’oggetto adattato: così facendo solo i metodi definiti nell’interfaccia HList sono visibili all’esterno.  
Inoltre il fatto di utilizzare l’Object adapter “obbliga” durante la scrittura del codice a scrivere sempre il nome dell’oggetto: in tal modo evito possibili errori dovuti alla mancanza della parola chiave “super” nell’invocazione dei metodi (che non viene segnalata come errore dal compilatore).

**Iterator Pattern:** Questo behavioral pattern permette l’utilizzo di un oggetto detto Iteratore per attraversare un contenitore e accedere ai suoi elementi senza conoscere i dettagli implementativi del contenitore stesso garantendo così uniformità di accesso.

**Arrange Act Assert (AAA):** E’ un pattern molto utilizzato nella scrittura dei test cases che permette di scrivere buoni test.  
Prevede l’utilizzo in ordine delle seguenti 3 operazioni:

* Arrange: Ha il compito di impostare le precondizioni del test case, ovvero settare gli oggetti nello stato corretto per poter poi eseguire il test.  
  (Nei test creati qualora questo non sia presente e’ perché tutta l’inizializzazione è già stata eseguita nel metodo segnato dal tag @Before).
* Act: Agisce sul comportamento desiderato. Questo step deve solo eseguire l’aspetto principale da testare.
* Assert: Ha il compito di verificare se ciò che si è fatto in Act ha prodotto il risultato aspettato o meno.

Si è scelto di utilizzare questo pattern in modo da rendere i test semplici e mirati su una particolare funzionalità rendendo il test indipendente dalle altre funzionalità definite nel programma.

Altra problematica che ha spinto ad utilizzare questo pattern adottando poche asserzioni per test case è la seguente: inserendo molte asserzioni in un test, qualora la prima fallisca, tutte le successive non vengono eseguite. Questo implica che nel caso in cui un test fallisca a volte possa risultare complicato capire cosa sia effettivamente andato storto.

Non è stato possibile adottare questo pattern per il controllo delle eccezioni in quanto prevede l’esecuzione del metodo all’interno del costrutto definito per l’asserzione. (si sono uniti act e assert)