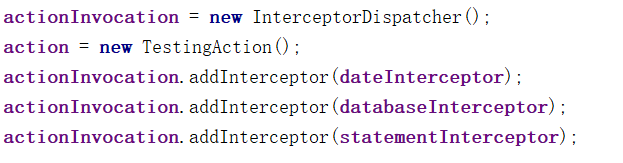
**New version with more code**

**Interceptor design pattern**

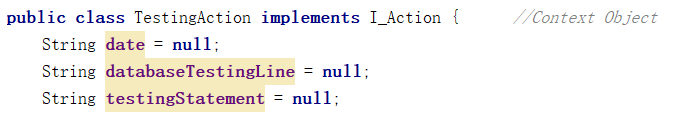
In the field of software development, an interceptor pattern is a software design pattern that is used when software systems or frameworks want to offer a way to change, or augment, their usual processing cycle.

Key aspects of the pattern are that the change is transparent and used automatically. In essence, the rest of the system does not have to know something has been added or changed and can keep working as before. To facilitate this, a predefined interface for extension has to be implemented, some kind of dispatching mechanism is required where interceptors are registered and context objects are provided, which allow access to the framework's internal state.

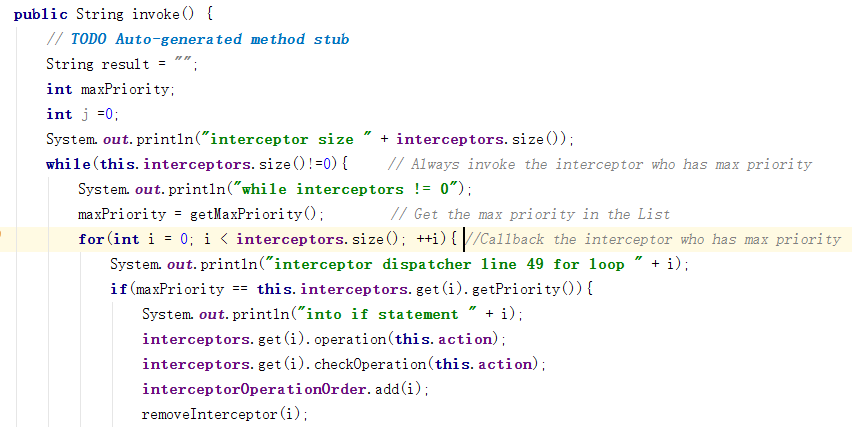
In our project, we use Interceptor design pattern to implement logger operation. There are three different interceptors - DateInterceptor (Automatic create log time), DatabaseTestingLineInterceptor (Check validity of Testing line) and TestingStatementInterceptor (Check validity of Testing result), these interceptors will be added in dispatcher as specific out-of-band services and they will use context object (action) to control the concrete framework.



TestingAction is know as context Object class. TestingAction allow access to framework’ internal state. For instance, a TestingAction object will get database testing information and testing result from testing framework which will be operated by different interceptors.



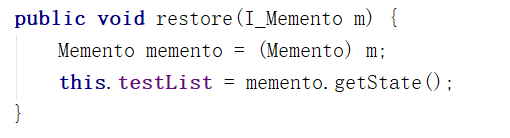
InterceptorDispatcher, which allows applications to register and remove concrete interceptor. InterceptorDispatcher use priority call back strategy - each interceptor have its own priority number. The dispatcher always invoke the interceptor who has the max priority number firstly .



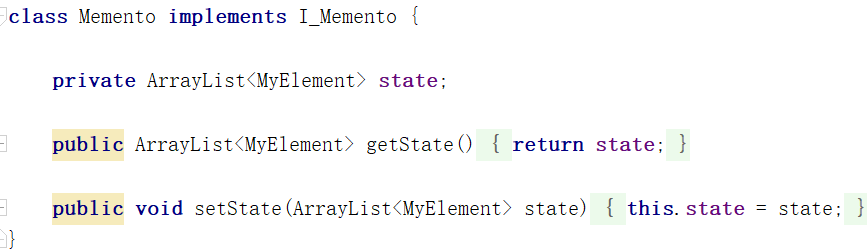
**Memento design pattern**

Memento design pattern can capture and externalise an object’s internal state so that the object can be restored to this state later without any violating encapsulation. In our project, the memento design pattern was used in undo operation in select XML/Json elements which be chosen as a test case of running test. This operation allow the user to ‘back out’ and recover form error operations.There are three important parts of memento design pattern in our project - DataOriginator (The ‘thing’ that ‘changes’), Caretaker (The ‘thing’ that changes the originator) and memento(The state of the originator before the change)

In DataOriginator, the restore method will get the previous state of Originator.

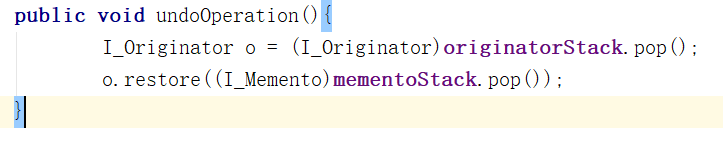


In Memento, it can store the current state of selection operation.

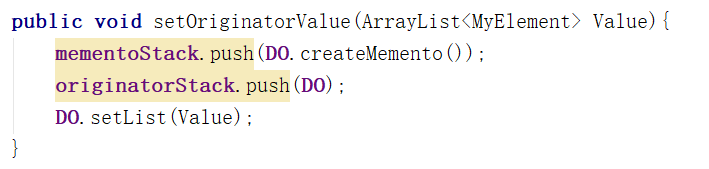


In Caretaker, there are two important methods: undoOperation() and setOriginatorValue().

undoOperation will get a previous state from stack and restore it which will make originator return to previous state.



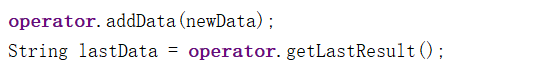
In setOriginatorValue(), the caretaker will create new memento to store the current state and push it into stack.

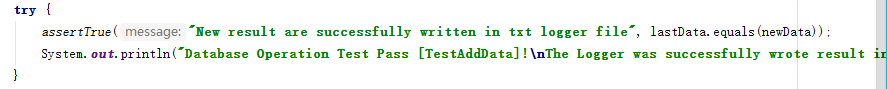


**Evidence of Testing**

For this project, we implemented a Test Primer which runs 2 Test Suits. First one is DatabaseOperationTest: JUnit testing for TestAddData, Second one is InterceptorDispatcherTest: JUnit testing for TestInvoke and TestLoggerResultForm. There tests would tell us whether our database access method(text file operation) were working as we intend and whether dispatcher will work as we designed.

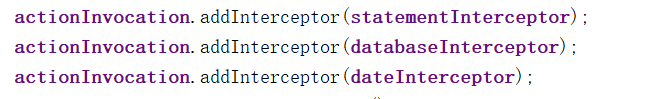
For DatabaseOperationTest: Testing of AddData, addData is a txt file operation method, which will get a new String type of testing result and write it into text file. In this JUnit test, test will pass once the new line is successfully added in and found as the last line.



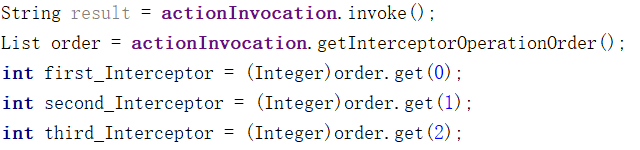


For InterceptorDispatcherTest: TestInvoke and TestLoggerResultForm, both of these methods are JUnit testing for Dispatcher.

For TestInvoke, it is a method which can check whether dispatcher will callback interceptors by priority strategy. As our project designed, each interceptor has its own priority, and the dispatcher will always invoke the interceptor who has the most large priority number currently. In this method, there different interceptors will be registered in dispatch in random ordering.

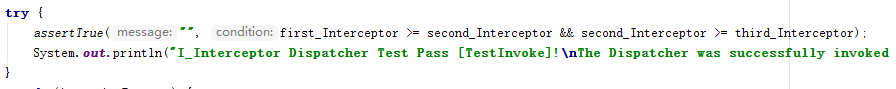


And it will check the callback ordering of three interceptors and theirs own priority.

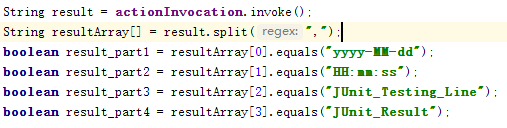


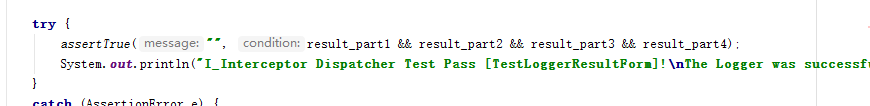
This JUnit testing will pass if the callback ordering is correct.

In this case, the correct result order is: 1.DateInterceptor 2. DatabaseInterceptor 3.StatementInterceptor.

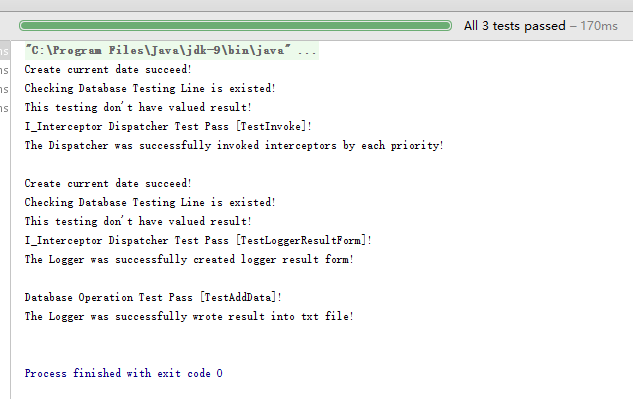


For TestLoggerResultForm, this method is designed to check whether dispatcher will return a right result form which will be written in log.txt. The JUnit testing will pass once the created form is “Date” + “Testing line” + “Testing Result”.





The test result as picture shows below. All 3 tests passed.



Add value priority of interceptor

In our project, we implement interceptor design pattern in logger operation and we use priority strategy as callback strategy. As our project designed, each interceptor has its own priority, and the dispatcher will always invoke the interceptor who has the most large priority number currently in the list. For example, the priority number of DateInterceptor is 3, if DateInterceptor has the max priority in the list, it will be callback firstly and remove from list after callback operation.

