## SOLID- Open Closed Principle

November 19th, 2011 [Mohamed Sanaulla](http://blog.sanaulla.info/author/MohamedSanaulla/) [Leave a comment](http://blog.sanaulla.info/2011/11/19/solid-open-closed-principle/#respond) [Go to comments](http://blog.sanaulla.info/2011/11/19/solid-open-closed-principle/#comments)

Open Closed Principle (OCP) states that,

**Software entities (Classes, modules, functions) should be OPEN for EXTENSION, CLOSED for MODIFICATION.**

Lets try to reflect on the above statement- software entities once written shouldn’t be modified to add new functionality, instead one has to extend the same to add new functionality. In otherwords you don’t touch the existing modules thereby not disturbing the existing functionality, instead you extend the modules to implement the new requirement. So your code is less rigid and fragile and also extensible.

OCP term was coined by [Bertnard Meyer](http://en.wikipedia.org/wiki/Bertrand_Meyer).

**How can we confirm to OCP principle?**

Its simple- Allow the modules (classes) to depend on the abstractions, there by new features can be added by creating new extensions of these abstractions.

**Let me try to explain with an example:**

Suppose you are writing a module to approve personal loans and before doing that you want to validate the personal information, code wise we can depict the situation as:

|  |  |  |
| --- | --- | --- |
| 01 | public class LoanApprovalHandler | |
| 02 | { |

|  |  |  |
| --- | --- | --- |
| 03 | public void approveLoan(PersonalValidator validator)<br /> | |
| 04 | { |

|  |  |  |
| --- | --- | --- |
| 05 | if ( validator.isValid()) | |
| 06 | { |

|  |  |  |
| --- | --- | --- |
| 07 | //Process the loan. | |
| 08 | } |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 | } |

|  |  |  |
| --- | --- | --- |
| 11 | public class PersonalLoanValidator | |
| 12 | { |

|  |  |  |
| --- | --- | --- |
| 13 | public boolean isValid() | |
| 14 | { |

|  |  |  |
| --- | --- | --- |
| 15 | //Validation logic | |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |

So far so good. As you all know the requirements are never the same and now its required to approve vehicle loans, consumer goods loans and what not. So one approach to solve this requirement is to:

|  |  |  |
| --- | --- | --- |
| 01 | public class LoanApprovalHandler | |
| 02 | { |

|  |  |  |
| --- | --- | --- |
| 03 | public void approvePersonalLoan (PersonalLoanValidator validator) | |
| 04 | { |

|  |  |  |
| --- | --- | --- |
| 05 | if ( validator.isValid()) | |
| 06 | { |

|  |  |  |
| --- | --- | --- |
| 07 | //Process the loan. | |
| 08 | } |

|  |  |
| --- | --- |
| 09 | } |
| 10 | public void approveVehicleLoan (VehicleLoanValidator validator ) | |

|  |  |
| --- | --- |
| 11 | { |
| 12 | if ( validator.isValid()) | |

|  |  |
| --- | --- |
| 13 | { |
| 14 | //Process the loan. | |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 | } |

|  |  |  |
| --- | --- | --- |
| 17 | // Method for approving other loans. | |
| 18 | } |

|  |  |  |
| --- | --- | --- |
| 19 | public class PersonalLoanValidator | |
| 20 | { |

|  |  |  |
| --- | --- | --- |
| 21 | public boolean isValid() | |
| 22 | { |

|  |  |  |
| --- | --- | --- |
| 23 | //Validation logic | |
| 24 | } |

|  |  |
| --- | --- |
| 25 | } |
| 26 | public class VehicleLoanValidator | |

|  |  |
| --- | --- |
| 27 | { |
| 28 | public boolean isValid() | |

|  |  |
| --- | --- |
| 29 | { |
| 30 | //Validation logic | |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 | } |

We have edited the existing class to accomodate the new requirements- in the process we ended up changing the name of the existing method and also adding new methods for different types of loan approval. This clearly violates the OCP. Lets try to implement the requirement in a different way

|  |  |
| --- | --- |
| 01 | /\*\* |
| 02 | \* Abstract Validator class | |

|  |  |
| --- | --- |
| 03 | \* Extended to add different |
| 04 | \* validators for different loan type | |

|  |  |
| --- | --- |
| 05 | \*/ |
| 06 | public abstract class Validator | |

|  |  |
| --- | --- |
| 07 | { |
| 08 | public boolean isValid(); | |

|  |  |
| --- | --- |
| 09 | } |
| 10 | /\*\* | |

|  |  |  |
| --- | --- | --- |
| 11 | \* Personal loan validator | |
| 12 | \*/ |

|  |  |  |
| --- | --- | --- |
| 13 | public class PersonalLoanValidator | |
| 14 | extends Validator |

|  |  |
| --- | --- |
| 15 | { |
| 16 | public boolean isValid() | |

|  |  |
| --- | --- |
| 17 | { |
| 18 | //Validation logic. | |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | /\* |
| 22 | \* Similarly any new type of validation can | |

|  |  |  |
| --- | --- | --- |
| 23 | \* be accommodated by creating a new subclass | |
| 24 | \* of Validator |

|  |  |
| --- | --- |
| 25 | \*/ |

Now using the above validators we can write a LoanApprovalHandler to use the Validator abstraction.

|  |  |  |
| --- | --- | --- |
| 01 | public class LoanApprovalHandler | |
| 02 | { |

|  |  |  |
| --- | --- | --- |
| 03 | public void approveLoan(Validator validator) | |
| 04 | { |

|  |  |  |
| --- | --- | --- |
| 05 | if ( validator.isValid()) | |
| 06 | { |

|  |  |  |
| --- | --- | --- |
| 07 | //Process the loan. | |
| 08 | } |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 | } |

So to accommodate any type of loan validators we would just have create a subclass of Validator and then pass it to the approveLoan method. That way the class is CLOSED for modification but OPEN for extension.

**Another example:**

I was thinking of another hypothetical situation where the use of OCP principle can be of use. The situation is some thing like: “We maintain a list of students with their marks, unique identification(uid) and also name. Then we provide an option to get the percentage in the form of uid-percentage name value pairs.”

|  |  |  |
| --- | --- | --- |
| 01 | class Student | |
| 02 | { |

|  |  |
| --- | --- |
| 03 | String name; |
| 04 | double percentage; | |

|  |  |
| --- | --- |
| 05 | int uid; |
| 06 | public Student(String name, double percentage, int uid) | |

|  |  |
| --- | --- |
| 07 | { |
| 08 | this.name = name; | |

|  |  |  |
| --- | --- | --- |
| 09 | this.percentage = percentage; | |
| 10 | this.uid = uid; |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 | } |

We collect the student list into a generic class-

|  |  |
| --- | --- |
| 01 | class StudentBatch { |
| 02 | private List<Student> studentList; | |

|  |  |
| --- | --- |
| 03 | public StudentBatch() { |
| 04 | studentList = new ArrayList<Student>(); | |

|  |  |
| --- | --- |
| 05 | } |
| 06 | public void getSutdentMarkMap(Hashtable<Integer, Double> studentMarkMap) { | |

|  |  |  |
| --- | --- | --- |
| 07 | if (studentMarkMap == null) { | |
| 08 | //Error |

|  |  |
| --- | --- |
| 09 | } else { |
| 10 | for (Student student : studentList) { | |

|  |  |  |
| --- | --- | --- |
| 11 | studentMarkMap.put(student.uid, student.percentage); | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| 14 | } |

|  |  |
| --- | --- |
| 15 | /\*\* |
| 16 | \* @param studentList the studentList to set | |

|  |  |
| --- | --- |
| 17 | \*/ |
| 18 | public void setStudentList(List<Student> studentList) { | |

|  |  |  |
| --- | --- | --- |
| 19 | this.studentList = studentList; | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

Suppose we need to maintain the order of elements in the Map by their insertion order, so we would have to write a new method to get the map in the insertion order and for that we would be using LinkedHashMap. Instead if the method- getStudentMarkMap() was dependent on the Map interface and not the Hashtable concrete implementation, we could have avoided changing the StudentBatch class and instead pass in an instance of LinkedHashMap.

|  |  |  |
| --- | --- | --- |
| 1 | public void getSutdentMarkMap(Map&lt;Integer, Double&gt; studentMarkMap) { | |
| 2 | if (studentMarkMap == null) { |

|  |  |  |
| --- | --- | --- |
| 3 | //Error | |
| 4 | } else { |

|  |  |
| --- | --- |
| 5 | for (Student student : studentList) { |
| 6 | studentMarkMap.put(student.uid, student.percentage); | |

|  |  |  |
| --- | --- | --- |
| 7 | } | |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |

PS: I know that Hashtable is an obsolete collection and not encouraged to be use. But I thought this would make another useful example for OCP principle.

Some ways to keep your code closer to confirming OCP:

* Making all member variables private so that the other parts of the code access them via the methods (getters) and not directly.
* Avoiding typecasts at runtime- This makes the code fragile and dependent on the classes under consideration, which means any new class might require editing the method to accommodate the cast for the new class.

Really good [article](http://www.objectmentor.com/resources/articles/ocp.pdf) written by Robert

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*