# Open-Closed Principle

***Software entities should be open for extension, but closed for modification***

## *Principle Foundation*

The heart of this principle addresses the most important truth associated with software development. System requirements will change and dictate the need for us to realize these changes in our software. The basis for this principle is quite simple actually. If we are able to change existing functionality within, or add new features to, our software system without having to modify our existing code base, the growth capability of our software increases significantly.

Many software systems exhibit a high degree of architectural rigidity. Seemingly simple changes often times are difficult to make, and are prone to the introduction of errors. Over time, the ramifications of many changes cause our code to rot, and the structural resiliency of the application is compromised. Eventually, systems with such rigid architectures become increasingly difficult to maintain, and tend to crumble. The Open-Closed Principle addresses this directly by allowing us to create flexible systems that are open for extension by adding new features without modifying our existing system classes, which are designed to be closed against change.

## *Sample Illustration*

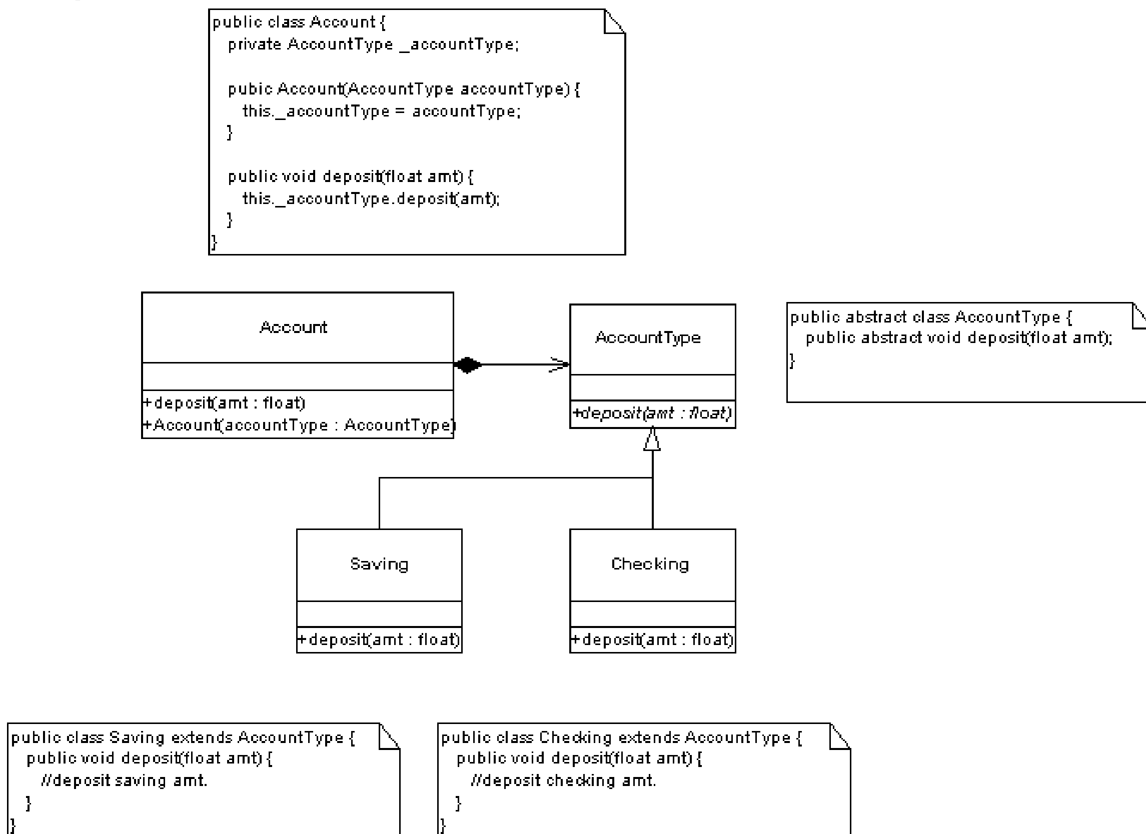


Figure 1

The diagram in Figure 1 illustrates how we can structure our application to accommodate extensibility. The realization of this flexibility is achieved through inheritance and polymorphism. In the above example, we see that our `Account` class has a relationship to an abstract `AccountType` class. Through dynamic binding, we are able to substitute descendents of `AccountType` anywhere that class is referenced. Subsequently, our `Account` class has no knowledge of the actual `Saving` and `Checking` concrete

classes, allowing us to create new `AccountType` descendents that our `Account` class may also use. These relationships are also illustrated in the accompanying code snippets.

## Key Implementation Considerations

- Abstract Coupling — To achieve closure, that aspect of the system must be bound to either an abstract class or interface. Binding classes at the concrete level implies changes must be made to alter behavior.
- Casting — A closed class cannot explicitly cast the abstract class or interface to which it is bound to an instance of any of the concrete classes. Nor should the closed class take advantage of RTTI (run-time type information) using the `instanceof` operator.
- Creation — To create an instance, the concrete class must be either explicitly referenced or loaded dynamically. A closed class, therefore, should not be responsible for creating concrete instances, instead deferring creation to a factory.
- System Closure — System wide closure is theoretically possible, but pragmatically undesirable. Achieving closure is complex, and should be judiciously applied to the areas of the system demanding this degree of flexibility. Major considerations include probability of change, likelihood of future growth, and complexity of a specific design challenge.

## Consequences

Achieving closure often times implies sacrificing simplicity for flexibility. Because the demands of closure dictate the creation of more classes, with complex structural and behavioral characteristics, the allocation of responsibilities becomes more challenging, yet significant. Simply put, a larger number of finer grained classes imply individual class responsibility must be very precise. Incorrectly assigning responsibilities negates the benefit of the flexible structure.

While possessing additional complexity, closure to modification with openness through extension reveals amazing advantages almost instantly. Systems are much easier to extend and maintain. Growth through extension is encouraged and embraced. Probably most important, however, is the resiliency that these systems exhibit over time. Their ability to accommodate change without compromising architectural integrity is staggering.

## Related Principles

Dependency Inversion Principle is the means through which Open-Closed Principle is achieved.

## References

[OOSC88] *Object-Oriented Software Construction.* Bertrand Meyer. Prentice-Hall, 1988.

[MARTIN00] *Design Principles and Design Patterns*. Robert C. Martin, 2000.

[GOF94] *Design Patterns: Element of Reusable Object-Oriented Software*. Gamma, Eric, et. al. Addison-Wesley, 1994.

[JOUP02] *Java Design: Objects, UML, and Process.* Kirk Knoernschild. Addison-Wesley, 2002.