# Dependency Inversion Principle

***Depend upon abstract entities, not concrete ones.***

## *Principle Foundation*

In object-oriented systems, there exists three degrees of coupling between classes; concrete, abstract, and nil. Concrete coupling exists when two classes, each concrete (or instantiable), directly reference each other. Nil coupling occurs between classes that have no direct relationship to each other. Last, abstract coupling is present when a concrete class has a relationship to another abstract class. Coupling at the abstract level allows for the greatest flexibility between two classes that must have a relation.

Dependency Inversion emphasizes flexible relationships between our system entities. Fundamentally, if we wish to comply with the Open-Closed Principle, then dependency inversion is the means through which compliance can be achieved.

## *Sample Illustration*

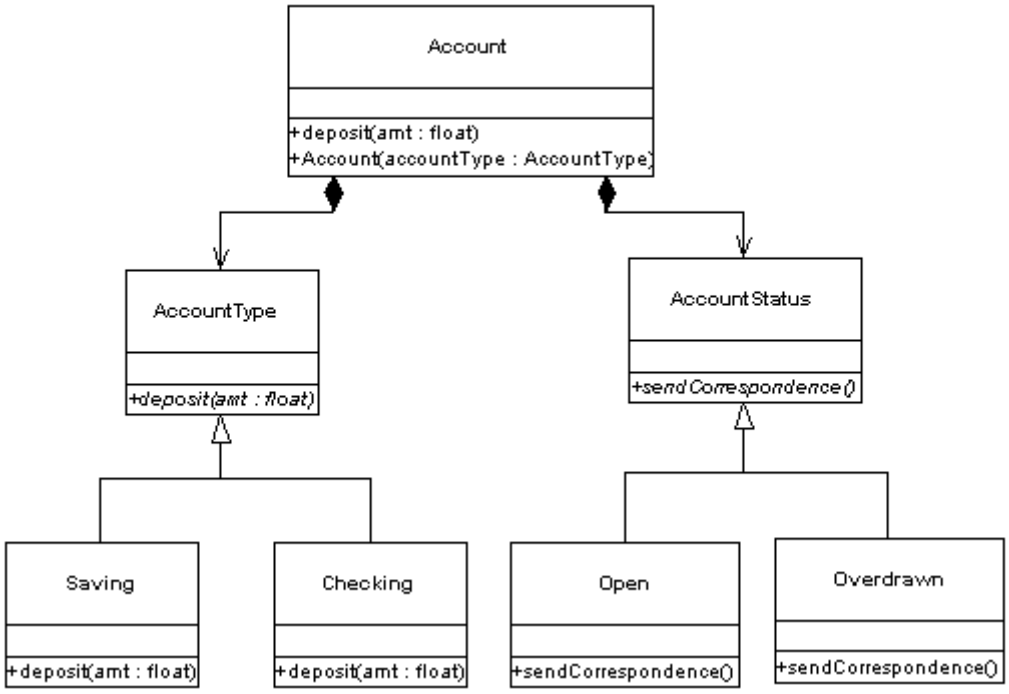


Figure 1

The diagram in Figure 1 shows that our classes are coupled at the abstract level. Doing so lends us great flexibility should we need to extend the system. As stated above, dependency inversion is the means through which we achieve compliance with the Open-Closed Principle.

## *Key Implementation Considerations*

- Abstract Coupling — Coupling abstractly is the key behind dependency inversion. Doing so allows substitution of descendents anywhere the ancestor is referenced. Caution must be exercised to free the client from any references to concrete descendents.
- Volatility — Coupling at the abstract level, while flexible, also offers additional complexities. In some situations, it may not be warranted because the likelihood of change is

 

low, or the behavior is very static. In these cases, depending on a concrete class may be desirable.

- 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.

## *Consequences*

While certainly powerful, dependency inversion can also be more challenging to implement. Because of the inverted nature of the relations, it's likely that object factories will need to be used for object creation to help avoid references to concrete instances.

We'll also find that this leads to a larger number of system classes. Therefore, while helping maintenance efforts by providing flexible application extension points, maintenance can be more difficult for those unfamiliar with object technology or the application architecture.

## *Related Principles*

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

Composite Reuse Principle

Stable Abstractions Principles and Stable Dependencies Principle use dependency inversion to create flexible package relationships.

## *References*

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

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

 