Dependency Inversion Principle

> High-level classes shouldn't depend on low-level classes. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions.

Usually when designing software, you can make a distinction between two levels of classes.

- **Low-level classes** implement basic operations such as working with a disk, transferring data over a network, connecting to a database, etc.
- **High-level classes** contain complex business logic that directs low-level classes to do something.

Sometimes people design low-level classes first and only then start working on high-level ones. This is very common when you start developing a prototype on a new system, and you're not even sure what's possible at the higher level because low-level stuff isn't yet implemented or clear. With such an approach business logic classes tend to become dependent on primitive low-level classes.

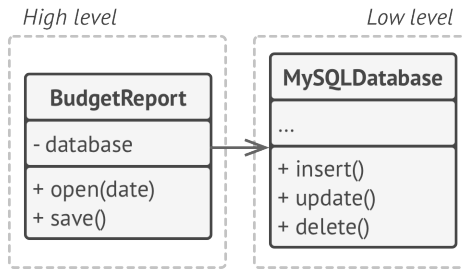
The dependency inversion principle suggests changing the direction of this dependency.

1. For starters, you need to describe interfaces for low-level operations that high-level classes rely on, preferably in business terms. For instance, business logic should call a method `openReport(file)` rather than a series of methods `openFile(x)`, `readBytes(n)`, `closeFile(x)`. These interfaces count as high-level ones.
2. Now you can make high-level classes dependent on those interfaces, instead of on concrete low-level classes. This dependency will be much softer than the original one.
3. Once low-level classes implement these interfaces, they become dependent on the business logic level, reversing the direction of the original dependency.

   The dependency inversion principle often goes along with the *open/closed principle*: you can extend low-level classes to use with different business logic classes without breaking existing classes.

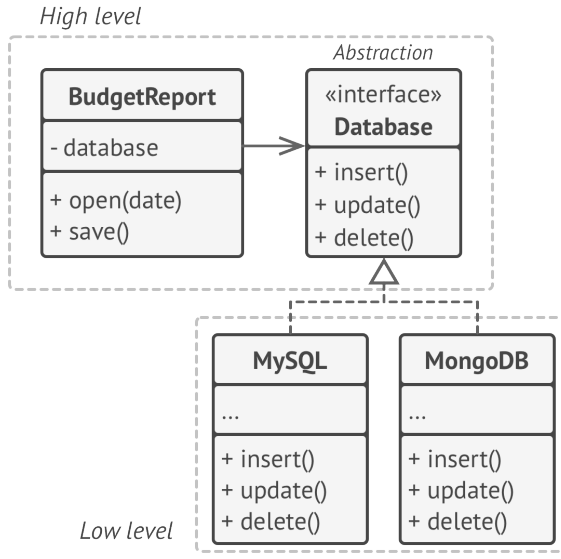
Example

In this example, the high-level budget reporting class uses a low-level database class for reading and persisting its data. This means that any change in the low-level class, such as when a new version of the database server gets released, may affect the high-level class, which isn't supposed to care about the data storage details.



BEFORE: a high-level class depends on a low-level class.

You can fix this problem by creating a high-level interface that describes read/write operations and making the reporting class use that interface instead of the low-level class. Then you can change or extend the original low-level class to implement the new read/write interface declared by the business logic.



AFTER: low-level classes depend on a high-level abstraction.

As a result, the direction of the original dependency has been inverted: low-level classes are now dependent on high-level abstractions.