SOLID

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Background

In the 1990's (and earlier), software developers noted that a distressing number of projects went badly or failed outright. Literature emerged examining reasons for this analysing why this was happening and what could be done to improve software development.

Among other things, practioners sought to identify a set of good development practices and patterns. One very influential article was *Design Principles and Design Patterns* by Robert C. Martin.

Bit rot

Martin noted that problems arise when software changes, either during initial development or later in its life. He called these problems “rot” and he identified various types of this rot.

Bit rot

Martin noted that problems arise when software changes, either during initial development or later in its life. He called these problems “rot” and he identified various causes this rot.

Causes of rot

Rigidity	Code that is difficult to change because changes to one part requires changes to many other parts
Fragility	Making a change at one point in the code causes it to break in other parts
Immobility	Modules can’t be moved into other projects or places within a project even when the intended function is the same.
Viscosity	It’s easier to make changes in a bad or hackish way than in a “good” way.

Most of these problems step from overly tight coupling of modules, or badly structured dependencies.

SOILD

Martin went on to identify five design principles that help make code resistent to rot. These have come to be known as the SOILD principles.

Single Responsibility Principle

A class should have only one reason to change.
Consider a class like this one:

```
class ReportGenerator
  read_data()
  calculate_results()
  write_report()
```

We might change this class if

1. the data sources change,
2. the formulas to calculate results change,
3. the report format changes.

This code is immobile. If we need to use the `calculate_results()` method in a different setting, this class isn't useful.

Single Responsibility Principle

We can fix this by splitting this into three classes

```
class DataReader

class ResultCalculator

class ReportWriter
```

Open-Closed Principle

A class should be open for extension but closed for modification.
Consider a class like this:

```
class DataRecord

    save_to_db()
      if db_type == MYSQL:
        # mysql-specific code
      else if db_type == POSTGRES:
        # postgres-specific code
      ...
```

Now suppose we want to add support for SQL Server.
This code is viscous, and probably also rigid.

Liskov Substitution Principle

Objects of a class should be replaceable by instances of subclasses. Suppose that we start our project using MySQL and build a MySQL interface class.

Later we decide to add support for Postgres, so we create a new class that inherits from the original MySQL class since it has the same behaviours. But there is never a case where we ask for a MySQL object and will be happy getting a Postgres one instead.

Liskov Substitution Principle

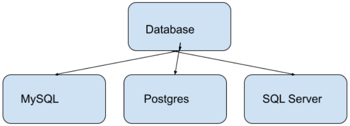
A better solution is to create a base `Database` class (probably abstract) and derive two child classes, one for MySQL and one for Postgres.

In a situation where the specific DBMS doesn't matter, we can specify the base `Database` class. If we get either of the child classes, then this will work. Note that we also need to write the child classes so that they satisfy the intent of the parent class.

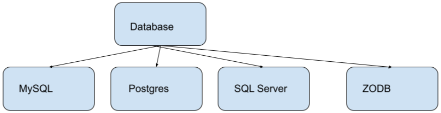
Interface Segregation Principle

A class should not depend on an interface it does not use.

Suppose we have a hierarchy of database interface classes.

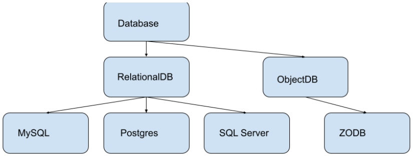


Now let's add some new types of database systems.



Interface Segregation Principle

The problem is that ZODB is an object database, while the previous classes all worked with relational databases. The base class may contain code that is specific to relational databases that ZODB does not use.



- Let `Database` define an interface common to all database systems.
- Let `RelationalDB` define the interface for relational databases.
- Let `ObjectDB` define the interface for object databases.

Dependency Inversion Principle

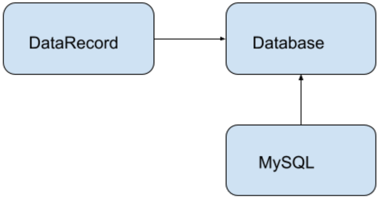
High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend on details. Details should depend on abstractions.

```
class DataRecord:

    def __init__(self):
        self.db = MySQL()
```

`DataRecord` is a high-level module. It shouldn't really matter what sort of database we use with it.

Dependency Inversion Principle



We fix this by defining an interface, `Database`, that tells `DataRecord` what it should expect and tells `MySQL` what it should provide.

Reference

https://fi.ort.edu.uy/innovaportal/file/2032/1/design_
principles.pdf