Database Systems Lab
Design Principles

Christian Rauch

August 26, 2024

SOLID

Key principles of object-oriented programming:

Single Responsibility Principle
Open/Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Adhering to these principles helps to decouple, modularize, replace, maintain, and extend components.

Single Responsibility Principle

A component should have only one job.

Examples:

- A complex system that handles HTTP requests, accesses the database, and has some business logic should have at least three components.
- JavaScript functions should ideally perform a single task, like ‘fetchProductData()‘ for calling the product API and ‘renderProductTable()‘ for updating the UI.
- Entity classes and business logic should be separated.
- Database repositories (data access logic) should be separate from service classes that implement business rules.

Open/Closed Principle

A component should be open for extension but closed for modification.

Examples:

- In a plugin-based architecture, new features can be added as plugins that extend the system without having the need to change other components.
- A payment processing system should allow new payment methods to be added by implementing a common interface.

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Examples:

- In your services, a 'Seller' class should be usable in place of a 'User' class, as the first only extends the functions of the latter.
- For a non-shippable 'DigitalProduct' (as subclass of 'Product') avoid overriding a method 'mark_as_shipped()' in a way that changes its expected behavior, e.g., returning None or raising an exception.

Interface Segregation Principle

Clients should only depend on interfaces they use.

Examples:

- Use fine-grained interfaces, e.g., ‘Shippable‘ and ‘Taxable‘, which can be selectively implemented by classes.
- A ‘Reviewable‘ interface could be implemented by classes like ‘Product‘ and ‘Seller‘, but not by others like ‘Order‘, ensuring only entities that can be reviewed have this capability.

Dependency Inversion Principle

High-level modules should only depend on abstractions of low-level modules.

Examples:

- The DB-API is an abstraction for concrete DB libraries.
- A high-level business-logic module ‘ShippingService‘ should depend on an interface ‘DataAccess‘ implemented by low-level modules like ‘SqliteDataAccess‘, ‘MariaDbDataAccess‘, ‘JsonFileDataAccess‘.

Dependency Registration

In modular systems you would usually use configuration to determine low-level depencies and often factory methods.

```
def create_da():
  if config["da.usefile"]:
    return JsonFileDataAccess()
  if config["da.usedb"]:
    if "maria" in config:
      return MariaDbDataAccess(
        config["da.db.details"])
    return SqliteDataAccess({"db_name": "lab1"})
  return InMemoryDataAccees()
```

These dependencies (or factories) are centrally registered (e.g., in app.py using the shared request-bound application context g).

```
from flask import Flask, g
@app.before_request
def before_request(): g.da_factory = create_da
# or: def before_request():   g.da = create_da()
```

High Level

High-level business logic components have low-level depencies.

```python
from flask import g

class ShippingService:
  def __init__(self):
    self.da = g.da_factory()
    self.mn = g.messenger_factory()
  def get_pending_orders(self, uid: int):
    self.da.read(...)
    self.get_order_details(...)
    ...
  def mark_shipped(self, oid: int, ...):
    self.da.insert('Message',
    {'msg': f'Shipped: {oid}', 'read': False})
    self.da.update('Order', oid, ...)
    mn.notify_customer(oid, 'shipped')
    self.da.read(...)
    self.da.update('Product', ...)
    ...
```

Abstractions

Abstract base for all implementations.

```python
from abc import ABC, abstractmethod

class DataAccess(ABC):
  @abstractmethod
  def insert(self, entity: str, attr: dict):
      pass
```

Abstract base for all Database-accessing implementations.

```python
class DbDataAccess(DataAccess):
  @abstractmethod
  def open_connection(): pass
  def __init__(self, con_params):
    self.con = self.open_connection(con_params)
  def __del__(self):
    if self.con: self.con.close()
  def insert(self, entity: str, attr: dict):
    cur = self.con.cursor()
    cur.execute("INSERT ...")
```

Low Level

Concrete implementations inherit from the base and implement what is missing.

```
import sqlite3
class SqliteDataAccess(DbDataAccess):
  def open_connection(self, con_params):
    return sqlite3.connect(con_params)
  ...

import mysql.connector
class MariaDbDataAccess(DbDataAccess):
  def open_connection(self, con_params):
    return mysql.connector.connect(con_params)
  ...

# query placholders differ etc.
```

Low Level

```python
import os
import json
from filelock import FileLock, Timeout

class JsonFileDataAccess(DataAccess):
    def __init__(self, dir: str): self.dir = dir
    def insert(self, entity: str, attr: dict):
        path = os.path.join(self.dir, f"{entity}.json")
        lock = FileLock(path, timeout=10)
        try:
            with lock:
                # TODO: robustness, error handling
                entities = json.load(path)
                entities.append(attr)
                with open(path, "w") as file:
                    json.dump(entities, file, indent='\t')
        except: ...
```

Conclusion

Thank you for your attention!