dependency inversion principle (DIP):

DIP’s formal definition:

1. High-level modules (Domain model or business logic) should not depend on low-level modules (infrastructure). Both should **depend on abstractions.**
2. Abstractions should not depend on details. Instead, details should depend on abstractions.

**High-level modules** are the code that your organization really cares about. Perhaps you work for a pharmaceutical company, and your high-level modules deal with patients and trials. Perhaps you work for a bank, and your high-level modules manage trades and exchanges. The high-level modules of a software system are the functions, classes, and packages that deal with our real-world concepts.

By contrast, **low-level modules** are the code that your organization doesn’t care about. It’s unlikely that your HR department gets excited about filesystems or network sockets. It’s not often that you discuss SMTP, HTTP, or AMQP with your finance team. For our nontechnical stakeholders, these low-level concepts aren’t interesting or relevant. All they care about is whether the high-level concepts work correctly. If payroll runs on time, your business is unlikely to care whether that’s a cron job or a transient function running on Kubernetes.

**Depends on doesn’t mean imports or calls, necessarily, but rather a more general idea that one module knows about or needs another module.**

Entity has a long living identity, it should be uniquely identifiable by using a reference (a uniquely identifiable variable) or by updating \_\_eq\_\_ and \_\_hash\_\_ methods

Value object is is short lived, defined by its attributes, if its values changes it is represented as a different object (identity crisis)

Service is just a function, not every unit has to be an entity or an value object

Business logic is also called as domain model

The **Repository pattern** is an abstraction over persistent storage

# Three layered architecture

Diagram

Description automatically generated

## What is domain model

Business logic layer is also called domain model, the domain is the fancy way of saying the problem you are trying to solve

## Difference between domain service and service layer

A **domain service (it is a piece of logic that belongs in the domain model)** represents a business concept or process, whereas a **service-layer service** represents a use case for your application.

Often the service layer will call a domain service

## What are ports and adapters

Ports and adapters came out of the OO world, and the definition we hold onto is that the port is the interface between our application and whatever it is we wish to abstract away, and the adapter is the implementation behind that interface or abstraction.

Now Python doesn’t have interfaces per se, so although it’s usually easy to identify an adapter, defining the port can be harder. If you’re using an abstract base class, that’s the port. If not, the port is just the duck type that your adapters conform to and that your core application expects—the function and method names in use, and their argument names and types.

Concretely, in this chapter, AbstractRepository is the port, and SqlAlchemyRepository and FakeRepository are the adapters.

Services.py - handles api response related logic as a mid way between url (urls.py) and views.py function

Accepting a request and sending response is handled In views.py