Onion Architecture

Ethan Kent

Spoonflower

July 28, 2022

AKA—

- Hexagonal Architecture
- Ports and Adapters
- Clean Architecture

Why? People have books to sell and blog posts to write, more or less?

Dependency Inversion Principle

Per Wikipedia:

1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Dependency Inversion Principle, continued

- Don’t refer to volatile concrete classes. Refer to abstract interfaces instead. This rule applies in all languages, whether statically or dynamically typed. . . .
- Don’t derive from volatile concrete classes. This is a corollary to the previous rule, but it bears special mention. . . .
- Never mention the name of anything concrete and volatile. This is really just a restatement of the principle itself.

Robert C. Martin, *Clean Architecture* 89 (2018).

Dependency Inversion Principle, continued

This is all pretty abstract. What are you actually saying?

Dependency Inversion Principle, continued

Let's say I have two concepts:

- A User, and
- an Express server with a `POST` endpoint called `updateUser`, which takes in a JSON payload and updates the PostgreSQL database to match the payload, and which server emits traces and metrics via Open Telemetry, and also checks a JWT bearer token to ensure the `admin` role is set properly.

Which is more stable, less likely to change, closest to our business domain, etc.?

Dependency Inversion Principle, continued

Should our User type include this data?

```
interface User {
  givenName: string;
  middleName?: string;
  familyName: string;
  sqlId: number;
  roleFromToken: string;
  oTelCorrelationId: string;
  isAuthorizedToUpdate: boolean;
}
```

Dependency Inversion Principle, continued

High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).

Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Dependency Inversion Principle, continued

Why?

> *Depend in the direction of stability.*
>
> *Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. . . . Some . . . components are designed to be volatile. We expect them to change.*
>
> *Any component that we expect to be volatile should not be depended on by a component that is difficult to change. Otherwise, the volatile component will also be difficult to change.*

Martin, *supra*, at 120.

Dependency Inversion Principle, continued

Let's apply this thinking to our `User` type.

- Are `Users` supposed to be volatile?
- Will `Users` stop having names?
- Will our application always use SQL?
- Would a business stakeholder recognize an Open Telemetry correlation ID as part of the concept of a User in the ubiquitous language of the business domain?

Dependency Inversion Principle, continued

In what way are we violating the Dependency Inversion Principle?

- A `User` is not supposed to be volatile, so it is the kind of thing that belongs in a low-level module.
- SQL, Open Telemetry, Express, Bearer Tokens, etc., are volatile, and so belong in high-level modules.
- Our `User`, in a low-level module, depends here on numerous high-level modules.

Dependency Inversion Principle, continued

Okay, this makes sense, but this whole “rely on abstraction not concretions” thing is a bunch of mumbo-jumbo.

Dependency Inversion Principle, continued

Fair enough.

Sometimes it will be necessary for lower-level modules to interact.

For example, it may be stable, non-volatile, fundamental to the business domain, etc. that—

- A User is authorized to do some things but not others.
- A User will be stored somewhere.
- The performance of the system will be monitored.

Dependency Inversion Principle, continued

We're going to be very careful to have a `User` type that is abstract and non-volatile, more or less in the very deepest, most core part of the application:

```
interface User {
  givenName: string;
  middleName?: string;
  familyName: string;
}
```

Sure the USER concept will be depended on by other, less general things, but the opposite shouldn't happen.

Dependency Inversion Principle, continued

But now we have a conflict:

- The idea of Logging is pretty abstract, but the implementation via, say, Pino or Winston is concrete.
- The idea of Authorization is pretty abstract, but the implementation via, say, an OAuth2-compatible Bearer Token is concrete.
- Within our abstract notion of Authorization, we may want to refer to the abstract notion of Logging, or
- Within our concrete implementation of Authorization via an OAuth2-compatible Bearer Token, we might want to refer to the abstract notion of Logging in order to avoid unnecessary coupling between different concerns.

Dependency Inversion Principle, continued

In short, we sometimes want something more abstract and stable to have the notion of some other concept that is more volatile. Perhaps our abstract concept of a Logger needs to depend on the details of an ID that comes from the concrete implementation of the abstract idea of Persistence.

Dependency Inversion Principle, continued

Let's pause for a second. Are you starting to get a mental image of layers here? And develop an instinct about which way things inside those layers should or should not know about one another?

Dependency Inversion Principle, continued

Back to the Dependency-Inversion Principle. How can a more abstract thing talk to a more concrete thing?

> *Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.*

Dependency Inversion Principle, continued

Define some abstractions:

```
interface AuthorizationService<Credential> {
  /** Authenticate a user asynchronously based on the credential. */
  authenticate: (credential: Credential) => Promise<boolean>;
}

interface PersistenceService<Id> {
  /** Get a user by ID. */
  getUser: (id: Id) => Promise<User>;
  /** Update a user, returning a `true` on success. */
  updateUser: (user: User) => Promise<boolean>;
}

interface LoggingService {
  /** Log a message synchronously. */
  log: (message: string) => void;
}
```

Dependency Inversion Principle, continued

Define some concretions (the details aren't important—what's important is that they are details):

```typescript
import jwksClient, { CertSigningKey, SigningKey } from "jwks-rsa";
import { type GetPublicKeyOrSecret, type VerifyOptions, verify, Secret, } from "jsonwebtoken";

// eslint-disable-next-line @typescript-eslint/no-non-null-assertion
const client = jwksClient({ jwksUri: process.env.JWKS_URI! });

const isCertSigningKey = (key: SigningKey | undefined): key is CertSigningKey =>
  !!key && Object.hasOwnProperty.call(key, "publicKey");

const getKey: GetPublicKeyOrSecret = (header, callback): void => {
  client.getSigningKey(header.kid, (_, key) => {
    const signingKey = isCertSigningKey(key) ? key.publicKey : key?.rsaPublicKey;
    callback(null, signingKey);
  });
};

const promiseVerify =
  (token: string, getKey: GetPublicKeyOrSecret | Secret, options: VerifyOptions): Promise<boolean> =>
    new Promise((resolve) =>
      verify(token, getKey, options, (err) => err ? resolve(false) : resolve(true))
    );

export const jwtAuthorizationService: AuthorizationService<string> = {
  authenticate: async (token: string) => promiseVerify(token, getKey, {}),
};
```

Dependency Inversion Principle, continued

Define some more concretions (the details aren't important—what's important is that they are details):

```
import { Client } from "pg";

const pgClient = new Client();

export const postgresPersistenceService: PersistenceService<number> = {
  getUser: async (id: number) => {
    pgClient.connect();

    const res = await pgClient.query("SELECT * FROM users WHERE id = $1", [id]);
    await pgClient.end();

    if (res.rowCount === 0) return null;
    if (res.rowCount > 1) throw new Error("Multiple users with the same ID");

    const { givenName, middleName, familyName } = res.rows[0];
    if (!givenName || !familyName) throw new Error("Invalid user");

    return { givenName, middleName, familyName };
  },

  // ...
```

Dependency Inversion Principle, continued

Define some more concretions (the details aren't important—what's important is that they are details):

```
  // ...

  insertUser: async (user: User) => {
    pgClient.connect();

    const { givenName, middleName, familyName } = user;

    try {
      await pgClient.query(
        "INSERT INTO users (given_name, middle_name, family_name) VALUES ($1, $2, $3) RETURNING id",
        [givenName, middleName, familyName]
      );

      await pgClient.end();
      return true;
    } catch (_) {
      return false;
    }
  },
};
```

Dependency Inversion Principle, continued

Define some more concretions (the details aren’t important—what’s important is that they are details):

```
import pinoCtor from "pino";

const pino = pinoCtor();

export const pinoLoggingService: LoggingService = {
  log: pino.debug,
};
```

Dependency Inversion Principle, continued

Okay, so what's the pattern here?

- The `User` interface is very abstract and depends on nothing else.
- The service interfaces (e.g. `LoggingService`) are fairly abstract and generic, and are designed to give our application exactly the functionality we want.
- The service implementations (e.g. `pinoLoggingService`) are very concrete—and kind of gross. They hide away all the tangled wires and allow us to use service interfaces that we designed for our application.

Dependency Inversion Principle, continued

Is this the same thing as Dependency *Injection*?