2022-04-17 Juhani Välimäki
Yrityksiä selittää muutamia Arkkitehtuuriluennon käsitteitä yleiskielellä.

DIP - Dependency Inversion Principle

Ylätason oliot eivät käytä suoraan alatason olioita, vaan käyttävät niiden abstraktioita. Esim. interface:a. Saadaan loose coupling ylätason olion ja jotakin sen tarvitsemaa toteuttavan alatason olion välille. Vaikkapa meillä olisi business-luokka, joka luulee käsittelevänsä Henkilöitä, mutta ajon aikana me annamme sille joko Opettajia tai Opiskelijoita, joilla on Henkilön piirteet ja joitakin omia piirteitä, ja osa Henkilön piirteistä voi olla toteutettu eri tavalla.

IoC - Inversion of Control

Normaalisti meidän koodi aloittaa ja ehkä kutsuu kirjastojen palveluita. IoC viittaa tilanteeseen, jossa framework kutsuukin meidän koodia. Tyypillistä esimerkiksi käyttöliittymäohjelmoinnissa, jossa selaimen tai käyttöjärjestelmän hiirtä ja näyttöä hallitseva järjestelmä saa tapahtumia (events) ja kutsuu sitten meidän tapahtumankäsittelijöitä. Eli me ei ollakaan sen isomman kuvan hallitsijoita, vaan ripustetaan meidän tapahtumankäsittelijäfunktioita tai -olioita (event-handlers) ja siten täytetään yksittäisiä aukkoja sovelluksen toiminnassa.

Dependency Injection

Ohjelmistosuunnittelun tekniikka, missä olio saa tarvitsemansa riippuvuuden/puuttuvan osan ulkoa päin asetettuna. Olio ei tiedä minkä toteutuksen saa, mutta jos se on toteutettu kuten tarkoitettu, olio osaa sen jälkeen toimia sitä hyödyntäen. Oliosta tulee tavallaan kokonainen. dependency injection -termejä:
"client" = Business-luokka (tarvitsee servicen, toteutuksen riippuvuudelleen)
"service" = Auto tai Moottoripyörä (toteutus clientin tarvitsemasta 'palvelusta'/ riippuvuudesta)
"injector" = DIPConfigurator (toimittaa/injektoi clientille sen tarvitseman riippuvuuden)

Loose coupling

Toteuttavaa osaa on helppo vaihtaa. Esimerkiksi webfrontin voi vaihtaa mobiilifrontiksi, jos sekin käyttää samaa backendiä (esim. REST API, GraphQL, ...) sillä käyttöliittymä ei riipu backendin sisäisestä toteutuksesta, vain APIsta. Tai tietokantatuotteen voi vaihtaa, helposti, jos backend on kirjoitettu fiksusti ja modulaarisesti ja esim kaikki asetukset luetaan jostakin .env-tiedostosta.

High cohesion

Moduulin/funktion kaikki osat kuuluvat yhteen, eikä niitä ole järkevä erottaa erilliseksi. Todennäköisesti siis toteuttaa Single-responsibility Principleä (SRP). Esimerkiksi kaikki yhtä Projekti-olion sisäistä tilaa/dataa käsittelevät funktiot laitetaan metodeiksi Projekti-luokkaan/olioon. Sen sijaa kaikki useata Projekti-oliota (esim. kokoelmaa) koskevat funktiot eivät kuulu sinne, vaan johonkin Projektien säilön tyyppiseen luokkaan. Ja taas jos meillä on joku liiketoimintaprosessi, joka riippuu yksittäisistä ja useista Projekteista, ja johon liittyy myös liiketoiminnan sääntöjä, menee kaikki prosesseihin liittyvät omaan luokkaansa.

Polymorphism ("Monimuotoisuus")

Esimerkiksi ohjelma luulee listaavansa Henkilöitä konsoliin, mutta oikeasti ajonaikaisesti siellä tulostuukin Opiskelijoita tai Opettajia sen mukaan, mitä tyyppiä ne Henkilö-listassa olevat oliot oikeasti ovat.

OCP - Open Closed Principle
- open for extension with new functionality modules
- closed for change (of already written, compiled, tested (and packed) solution/module)
- simple example: array.sort() function has been written already, not to be changed.
  BUT it can be extended by giving it a new ordering algorithm / function that we need.