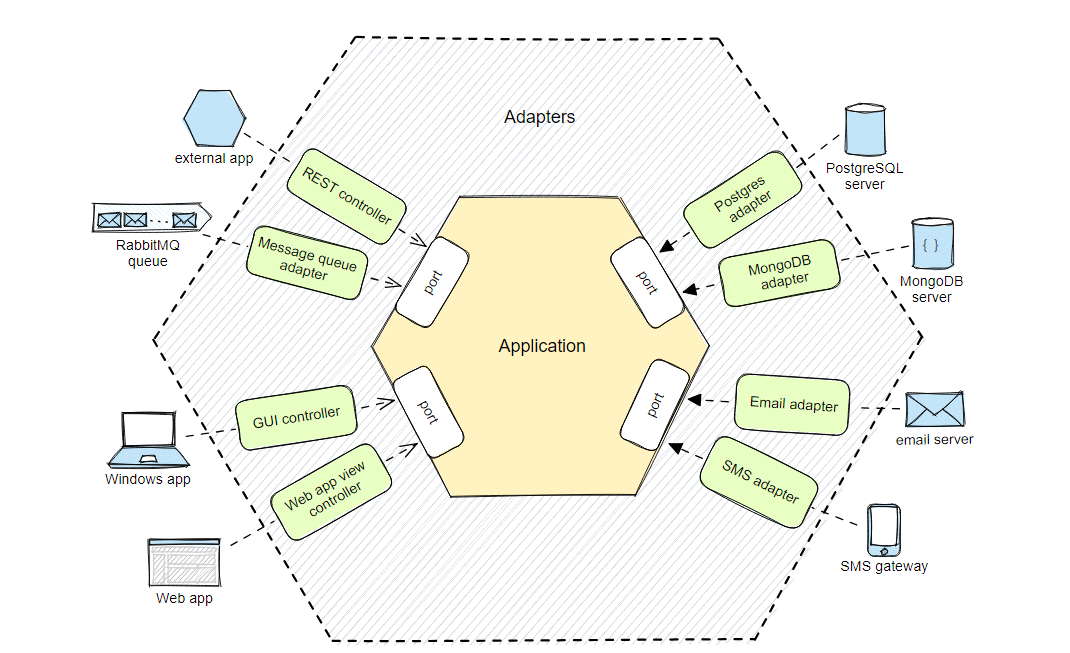
**Structura**

* **Hexagonal architecture** este un pattern pentru a separa bussiness logic de infrastructure, adica de a separa bussiness logic de mediul extern al apicatiei, ca si cum separam interiorul de exterior.
* Rolul la acest pattern este de a crea un component central inconjurat de interfete prin care poate fi accesat. Deci, multe se vor face prin interfete.
* **Mediul intern** - core logic
* **Mediul extern** - UI, messaging queue, database
* Acest principiu duce la aceea ca aplicatia poate fi controlata in mod egal de useri, alte programe sau teste,
* Rolul ei este de a structura aplicatia in asa fel incat sa putem testa si dezvolta aplicatia in isolatie, fara a apela la alte tehnologii externe ale ei(ca baza de date, 3th party REST API, frameworks etc.) si sa fie usor de configurat si de facut modificare, adica sa fie maintenable
* E formata din:

- **ports(interface layer)**

**- adaptors**

**- application/domain layer**



* Comunicarea dintre mediul intern si extern se face prin ports si adapters

**Ports**

* Ports reprezinta layer ce defineste interfetele ce stau la baza comunicarii dintre aplicatiei si systemului extern sau clienti
* Aplicatia comnica cu lumea externa anume prin porturi(interfete).Deci fac conexiunea dinte intern si extern.Porturile descriu scopul comunicatiei dintre aplicatie si lumea externa si se asigura sa lucreze bussiness logic code.
* Orice port poate avea mai multe adaptere in spatele sau.
* Porturile pot fi de 2 feluri:

- **Inbound**(de sosire) port - e ca un service interface care expune core logic lumii externe.(datele intra prin el in app)

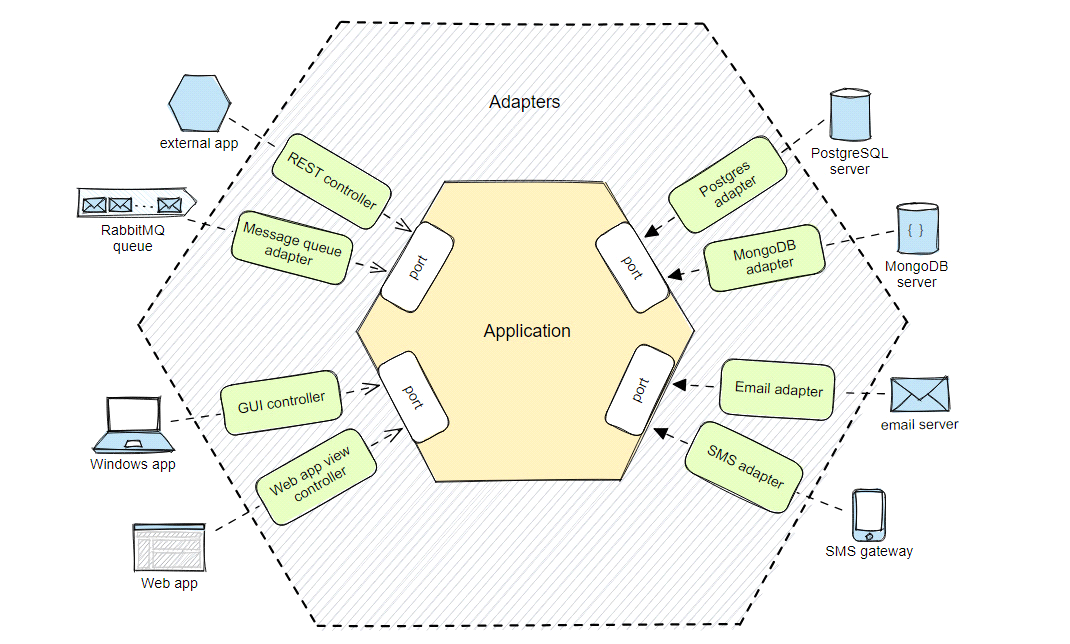
- **Outbound**(de plecare) port - e ca un repository interface care faciliteaza comunicatia de la aplicatie catre persistence system(datele ies din aplicatie)

**Adapters**

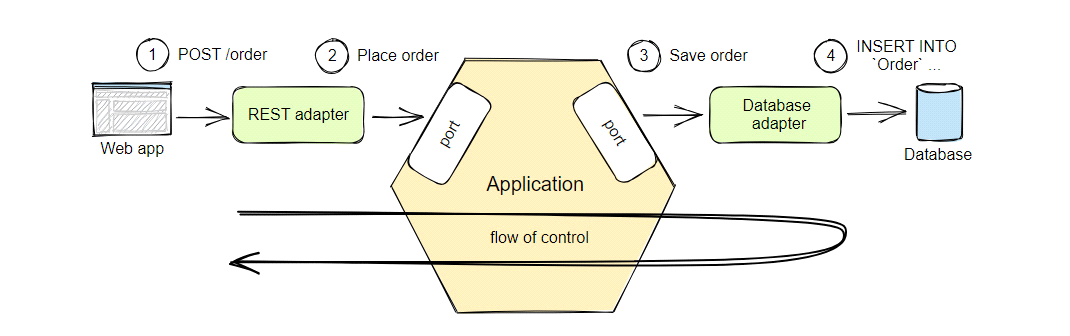
* **Layer ce contine implementarile interfetelor din ports layer**
* **Sunt implementarile porturilor care se ocupa de user input si le transforma in limbaj clar pentru un call.**
* Adapterele ofera conexiunea cu lumea externa, deci formeaza conexiunea dintre extern si intern. Ele transforma semnalele din lumea externa intr-un limbaj pe care aplicatia sa il inteleaga si invers
* Adapterele sunt interschimbabile fata de ambele parti fara a atinge bussiness logic. Asta face usor de a implementa cu timpul noi interfete si a folosi noi tehnologii
* **Tipuri de adapters:**

- **Primary**: adapters care controleaza aplicatia(desenate la stanga), folosind inbound port implementation, ca de ex RestControllers

- **Secondary**: adapters care sunt controlate de aplicatie(desenate la dreapta), adica aplicatia ia de la ele rapsunsuri. Sunt implementari la outbound port care face conexiuni cu messaging queues, database etc.



**Exemplu**

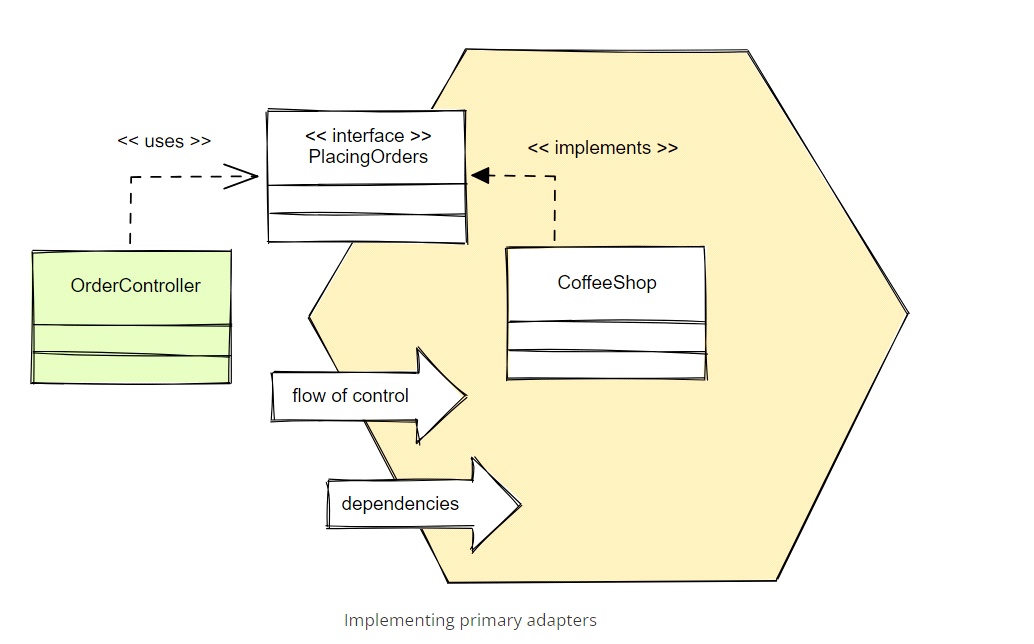


* De ex, in a coffee shop, este un punct de vanzare care se face prin UI, Cand se trimite un order(comanda), un REST adapter ia HTTP POST request si il traduce asa ca sa fie inteles de un port. Apelarea porturilor declasneaza bussiness logica legata de inserarea comenzii in aplicatie. Aplicatia nu stie ce este folosit in REST API
* Pe de alta parte, aplicatia comunica cu un port care permite salvarea comenzilor. Daca vrem gen sa lucram cu o baza relationara, un database adapter trebuie sa implementeze conexiunea cu baza de date. Adapterul ia informatia venita de la port si o transforma in SQL pentru a salva informatia(comenzile) in baza de date. Aplicatia nu stie ce tehnologie se implementeaza sau cum se utilizeaza ea.

**Implementation**

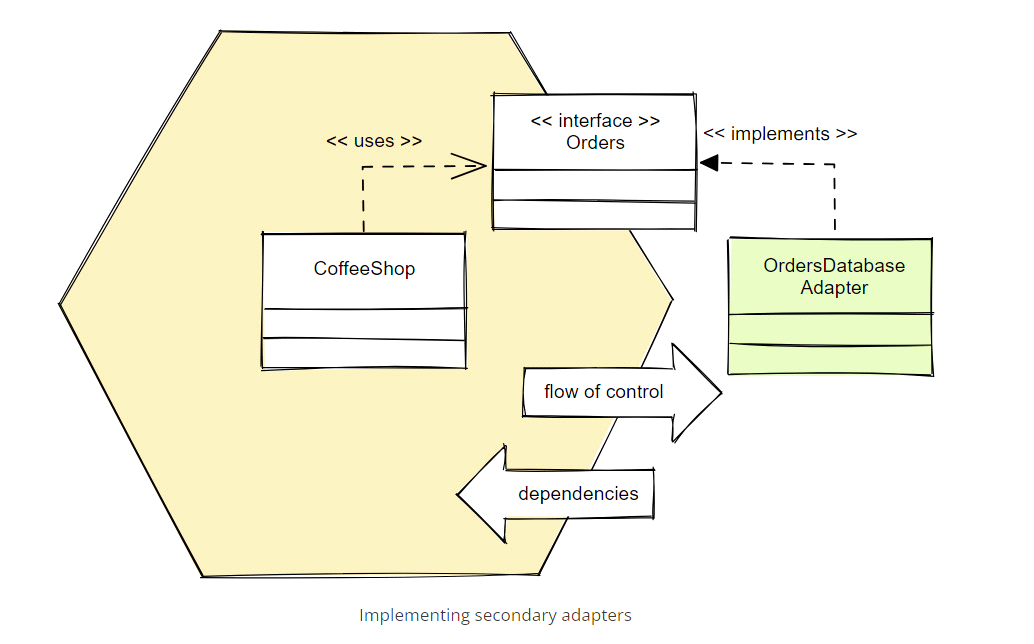
* **Primary adapters:**

Cand implementam un primary adapter, acesta trebuie sa spuna aplicatiei sa faca ceva. Controlul vine de la adapter la aplicatie prin ports.



Aici, OrderController este un adapter care apeleaza un caz(metoda) concreta din PlacingOrders(interface) port. In aplicatie, CoffeShop e clasa care implementeaza metodele portului sau mai bine zis a interfetei. Aplicatia habar nu are cine apeleaza functiunile sale.

* **Secondary adapter:**



Cand implementam secondary adapter, ciclul de control iese din aplicatie, caci vrem sa lasam database adapter sa stie ca el trebuie sa salveze date. Deoarece aplicatia nu trebuie sa stie nimic despre detaliile din lumea externa, folosim interfata tot. Ea nu trebuie sa fie dependenta de database adapter, de aceea ea foloseste un port, si adapter trebuie sa implementeze acest port.

Deci, CoffeeShop nu depinde de OrderDatabaseAdapter implementation direct, dar foloseste Order interface si lasa OrderDataVaseAdapter sa implementeze aceasta interfata,

**Layers**

* **Domain** - core bussiness logic layer, adica logica/codul de baza al aplicatiei. Implementarile la outer layers sunt ascuse de el
* **Application** - mediatori intre Domain si Framework layer
* **Framework** - are toate detaliile de implementare despre cum un domain layer va interactiona cu lumea externa

**Exemple**

* Domain - core al aplicatiei. Cake class e clasa in jurul careia domain logic lucreaza

public class Cake implements Serializable {

private static final long serialVersionUID

= 100000000L;

private String name;

// Getters and setters for the name

public String getName()

{

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString()

{

return "Cake [name=" + name + "]";

}

}

* Inbound port - interfata prin care core application comunica.

public interface CakeService {

public void createCake(Cake cake);

public Cake getCake(String cakeName);

public List<Cake> listCake();

}

* Outbound port: una sau mai multe interfete pentru a accesa outside world, ca Cake:

import java.util.List;

// Interface to access the cake

public interface CakeRepository {

public void createCake(Cake cake);

public Cake getCake(String cakeName);

public List<Cake> getAllCake();

}

* Primary Adapter: controller ce ofera endpoints pentru a crea si lua resurse:

@RestController

@RequestMapping("/cake")

public class CakeRestController implements CakeRestUI {

@Autowired

private CakeService cakeService;

@Override

public void createCake(Cake cake)

{

cakeService.createCake(cake);

}

@Override

public Cake getCake(String cakeName)

{

return cakeService.getCake(cakeName);

}

@Override

public List<Cake> listCake()

{

return cakeService.listCake();

}

}

* Secondary Adapter - implemnetatie la un outbound port:

@Repository

public class CakeRepositoryImpl

implements CakeRepository {

private Map<String, Cake> cakeStore

= new HashMap<String, Cake>();

@Override

public void createCake(Cake cake)

{

cakeStore.put(cake.getName(), cake);

}

@Override

public Cake getCake(String cakeName)

{

return cakeStore.get(cakeName);

}

@Override

public List<Cake> getAllCake()

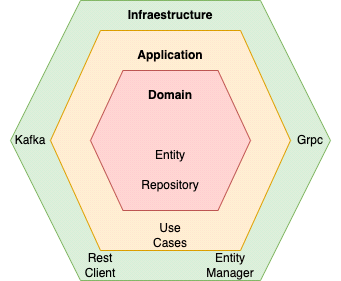
{

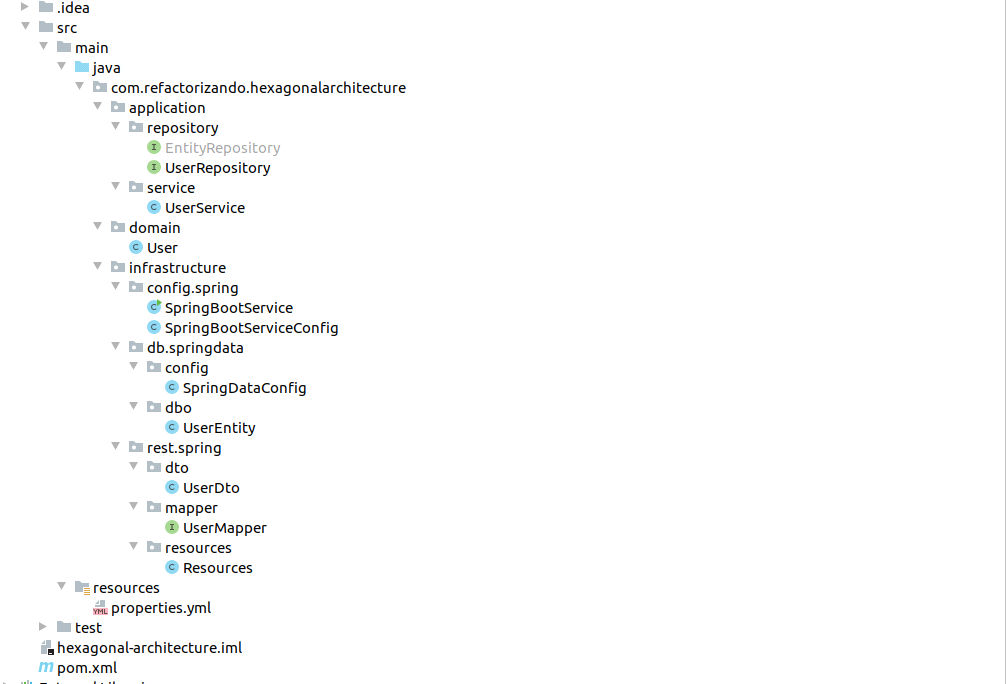
return cakeStore.values().stream().collect(Collectors.toList());

}

}

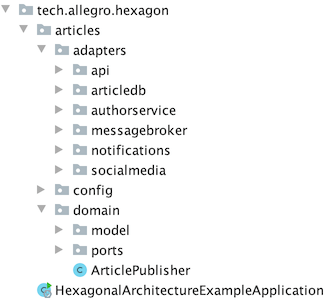
**Implementare**





* **Domain layer** - codul care implementeaza bussiness logic. Core al aplicatiei. Acest layer trebuie isolat de application layer si infrastucture layer. El comunica cu ele doar prin interfete.
* **Application Layer = ports** - punem aici tot ceea cu ce userul interactioneaza cu aplicatia, si anume interfetele doar
* **Infrastucture layer = adapters** - contine orice altceva aplicatia are nevoie, ca database configuration sau spring configuration sau ceea cu userul interactioneaza cu ea, ca RestController. Totodata, implementeaza interfetele toate interfetele folosite de domain din ports

**Structure of package**



* **domain** – este core al aplicatiei, si contien bussiness logic. Este independent de orice framework sau tegnologie. In el se definesc ports(interfetele) cu care lucreaza, dar care vor fi implementate in adapters
* **adapters** – sunt external API’s, si contin implentari la interfetele din domain
* **ports** – interfetele cu care lucreaza domain, si nu il intereseaza cum ele vor fi implementate si de cine.
* **API** – este usa de comunicare cu aplicatia. aici se pun API, ca REST CONTROLLER
* 
* 