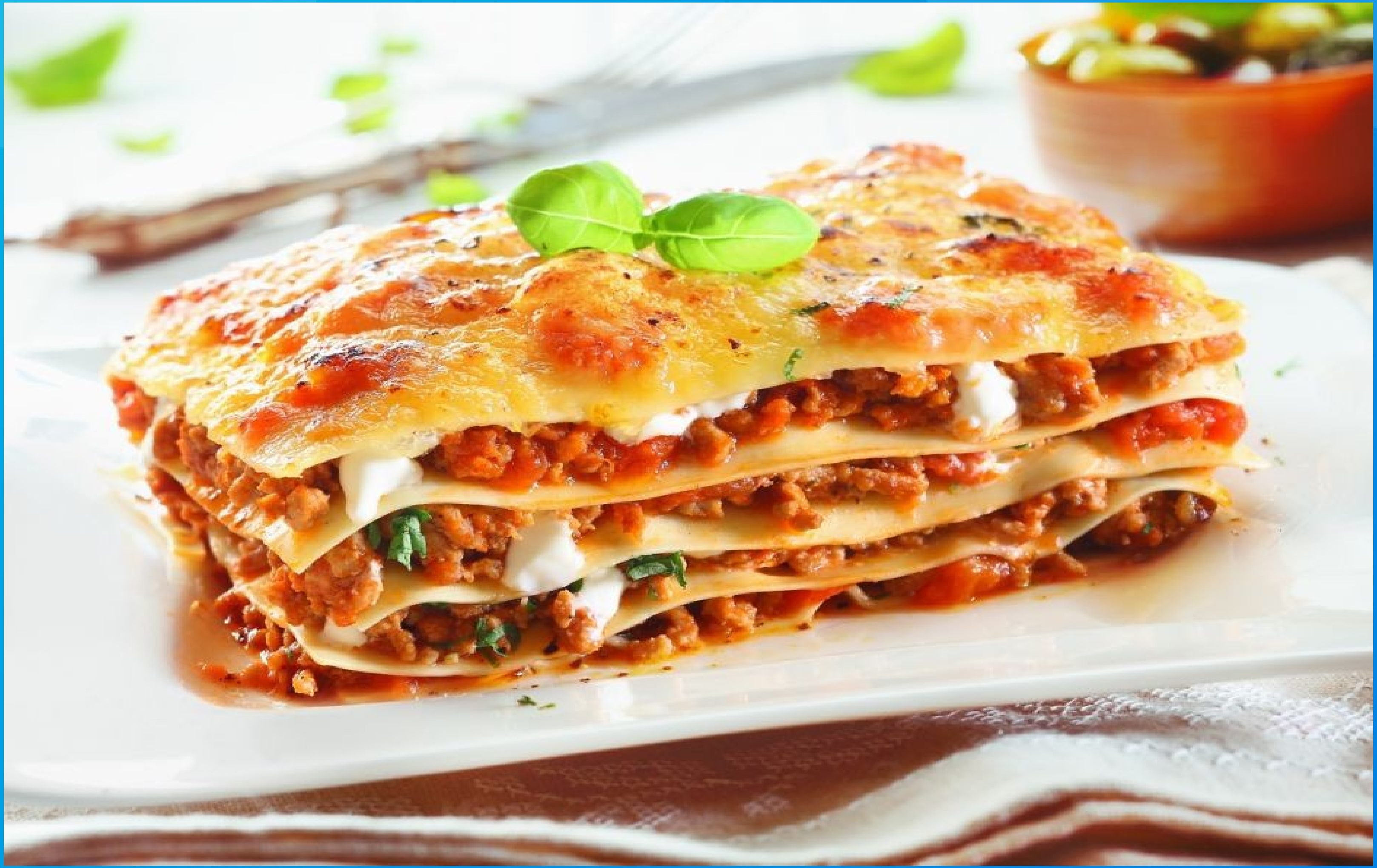
Hexagonal Architecture

Mateusz Winnicki

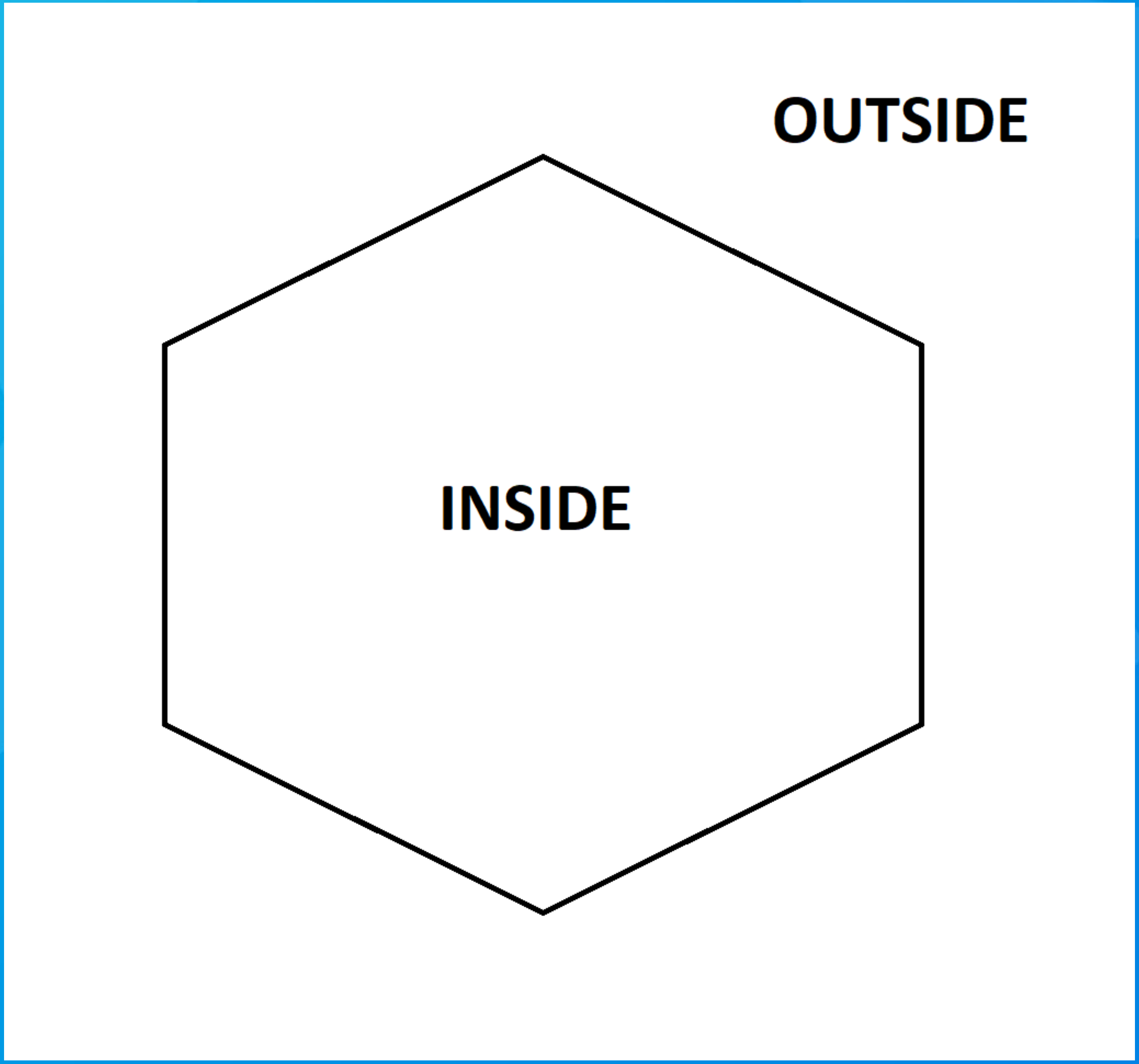
www.mateuszwinnicki.pl
mateusz.winnicki@euvic.pl

Sometimes we have

And sometimes we have



Hexagon

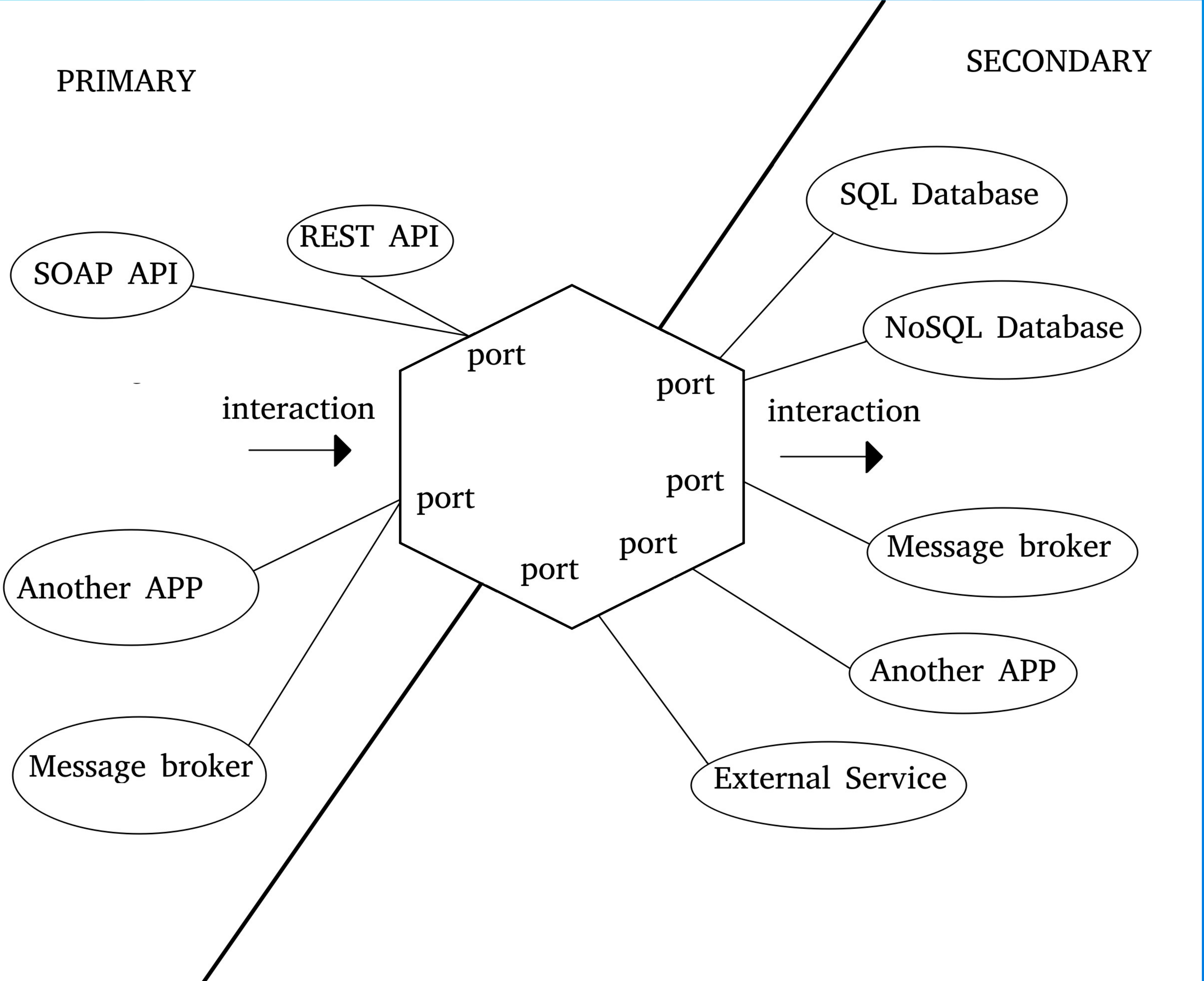


Inside

- **Domain**
- C**ontracts (aka ports)**
- Technology agnostic
- High isolation

Outside

- **Contracts implementations (aka Adapters)**
- Framework
- Environment
- Users
- Another hexagon?


PRIMARY
SECONDARY
REST API
SOAP API
SQL Database
NoSQL Database
port
port
interaction
interaction
port
port
port
port
Another APP
Message broker
Message broker
Another APP
External Service

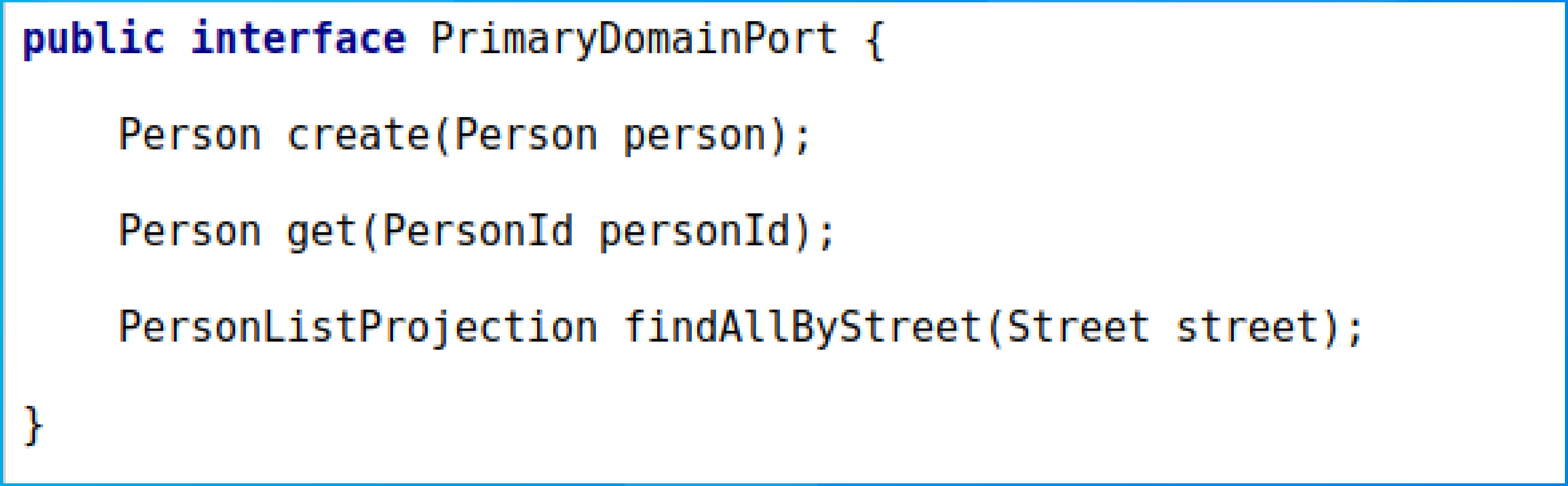
Primary and secondary?

Port

Port

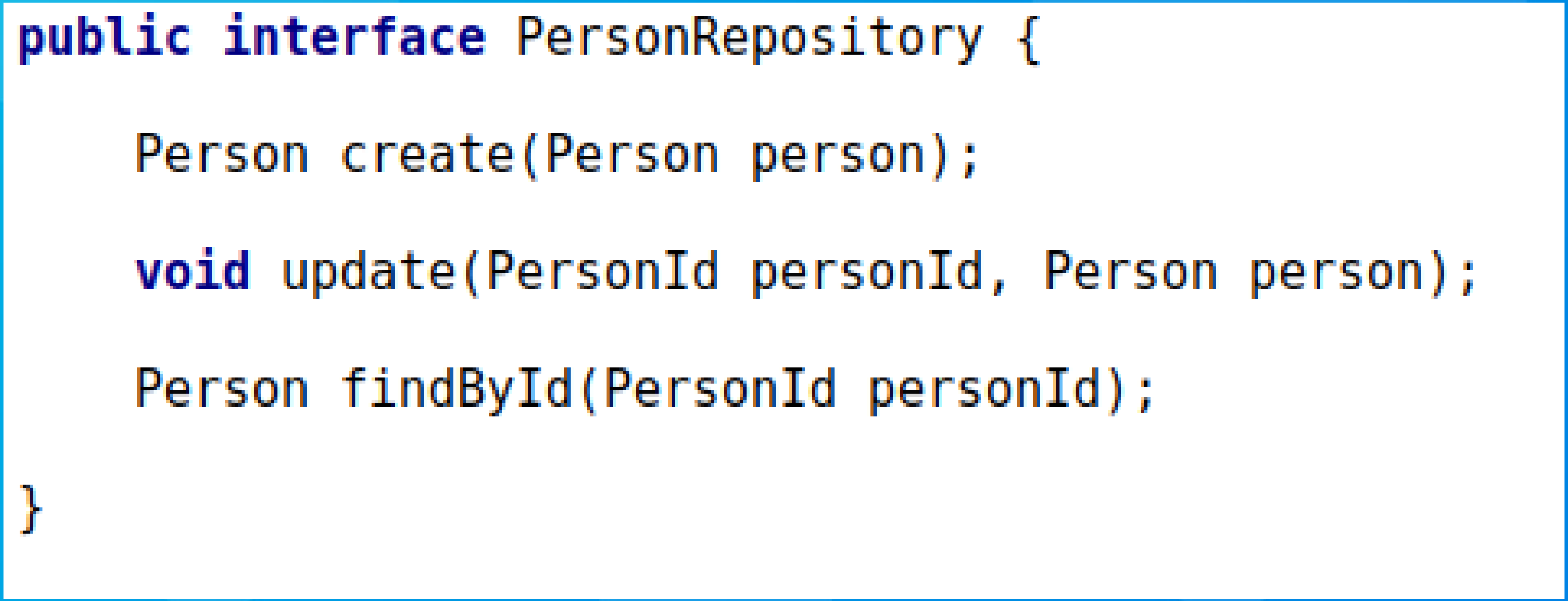
- Contract
- Defines how we can communicate with our domain (**primary**)
- Defines what our domain wants from the outside world (**secondary**)
- Should be named after interaction not by a technology behind
- Belongs to the domain

Primary port example



```
public interface PrimaryDomainPort {

    Person create(Person person);

    Person get(PersonId personId);

    PersonListProjection findAllByStreet(Street street);

}
```

Secondary port example

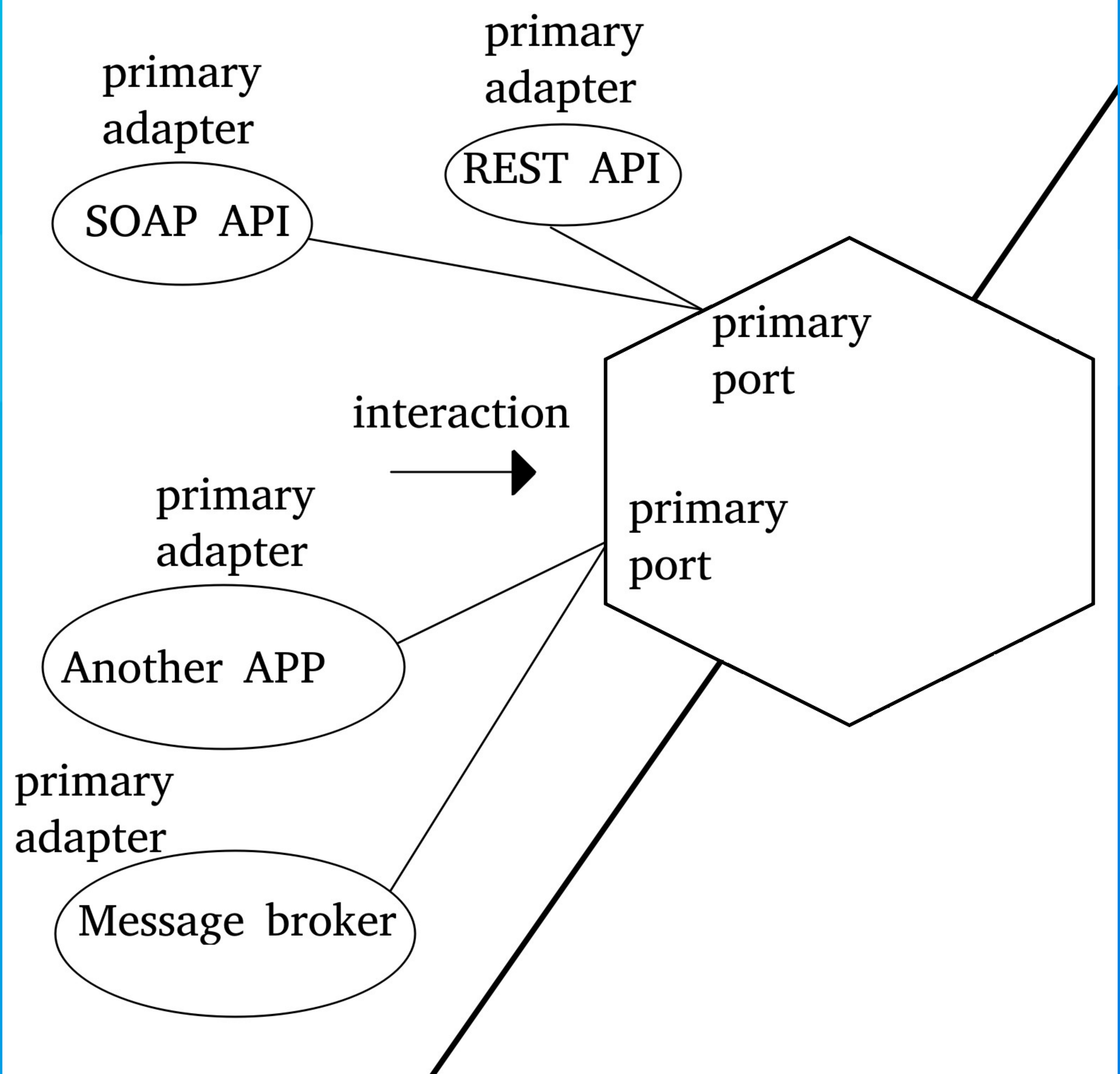


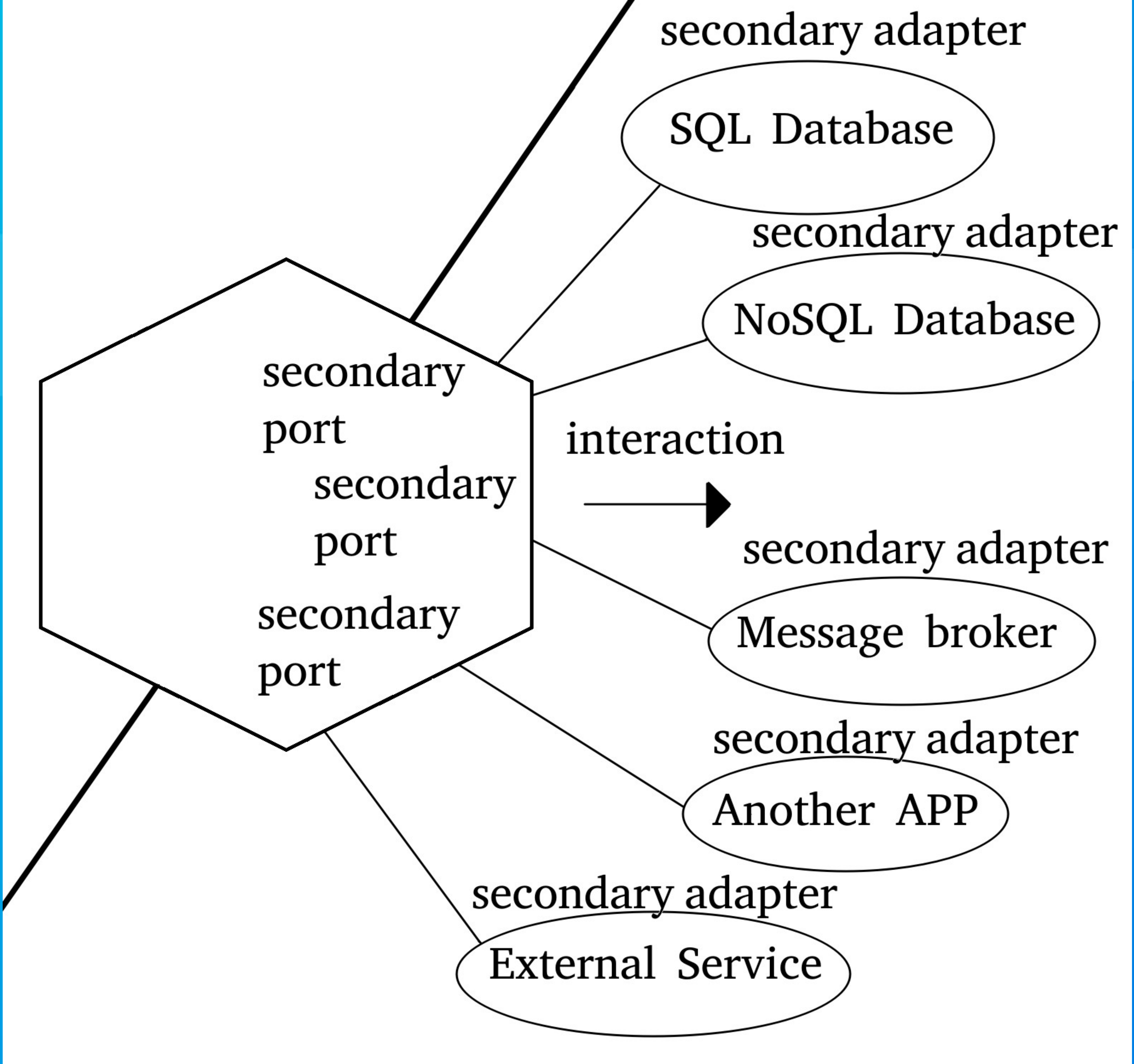
```
public interface PersonRepository {

    Person create(Person person);

    void update(PersonId personId, Person person);

    Person findById(PersonId personId);

}
```

Adapter

Adapter

- Contract’s implementation
- Translating technology request to agnostic request which can communicate with hexagon port (**primary**)
- Translating technology agnostic methods of the port to technology specific request
- Belongs to outside world

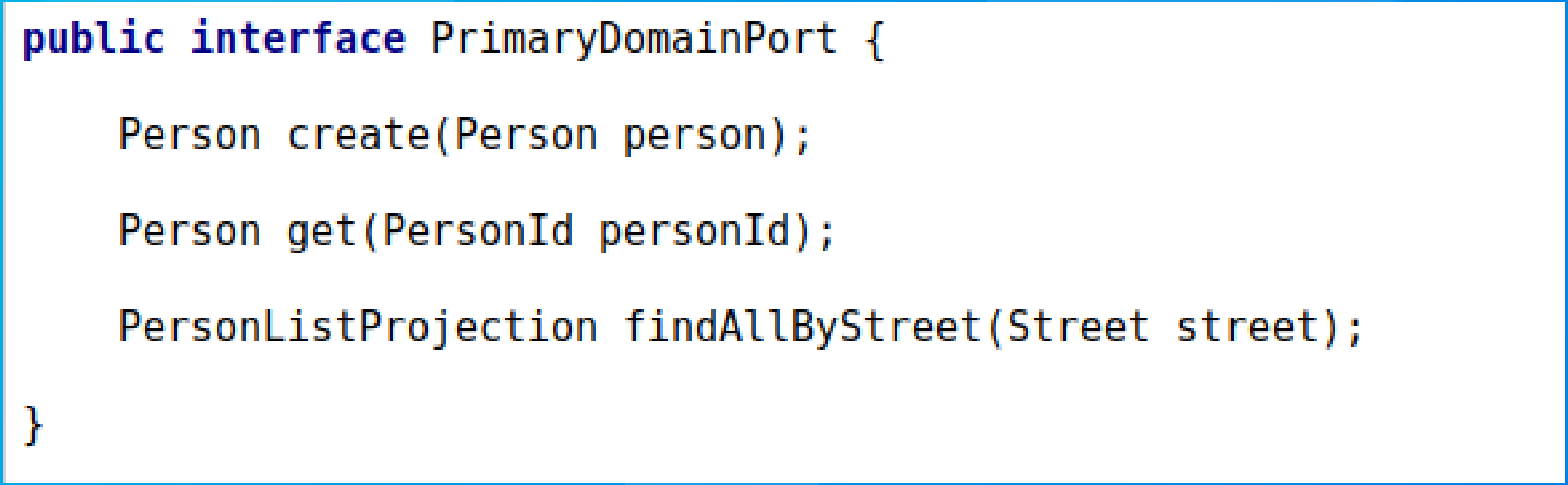

primary
adapter
SOAP API
primary
adapter
REST API
primary
port
interaction
primary
adapter
primary
port
Another APP
primary
adapter
Message broker


secondary adapter
SQL Database
secondary adapter
NoSQL Database
secondary port
secondary port
secondary port
interaction
secondary adapter
Message broker
secondary adapter
Another APP
secondary adapter
External Service

Main component aka Composition Root

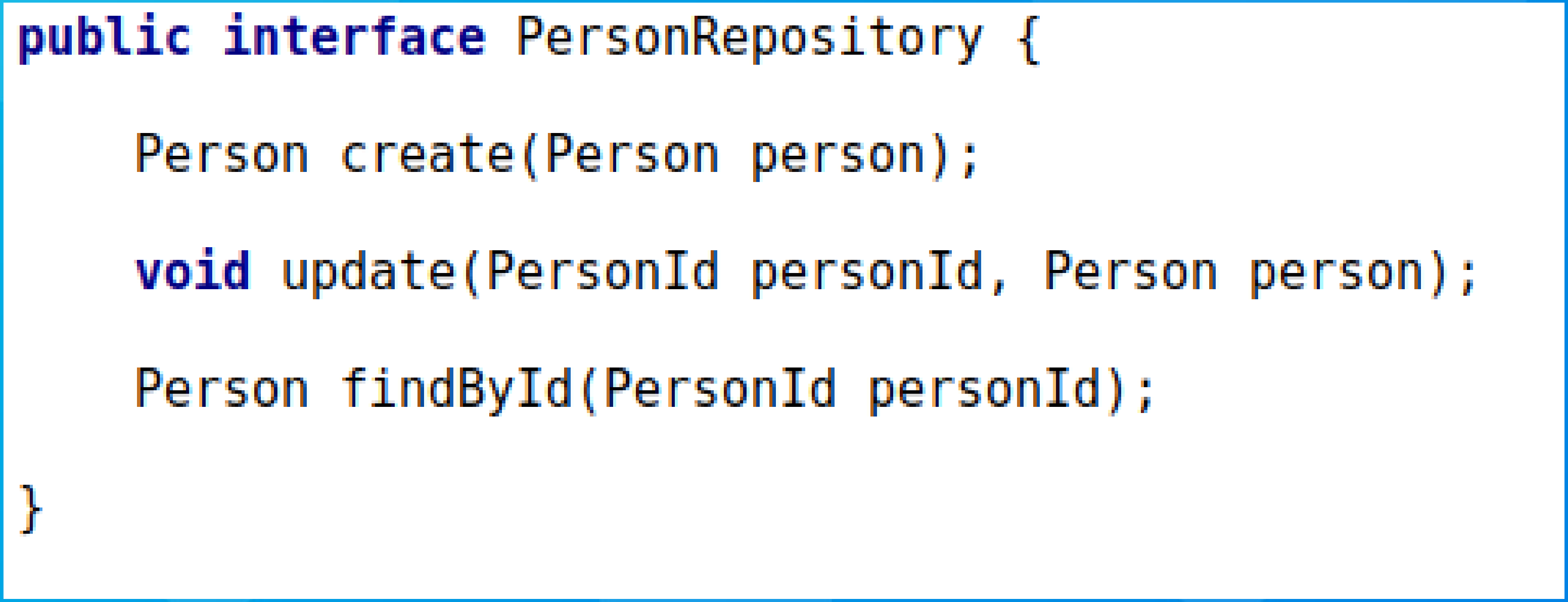
Let’s create hexagonal application! Recipe

1. Define what your domain will do – you will have your primary port (or ports)



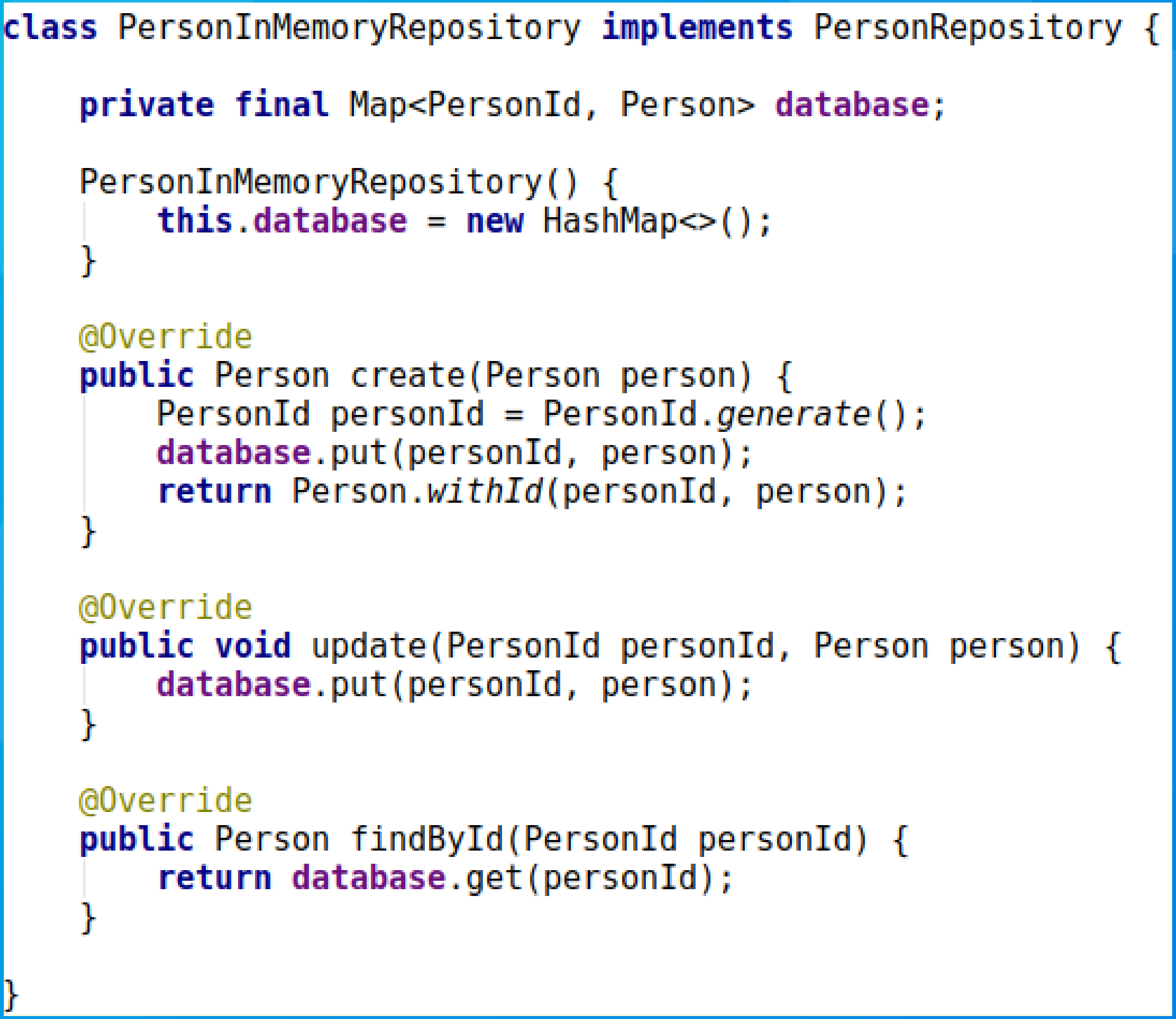
```java
public interface PrimaryDomainPort {

    Person create(Person person);

    Person get(PersonId personId);

    PersonListProjection findAllByStreet(Street street);

}
```

2. Define what your domain want – you will have your secondary ports (or one port)



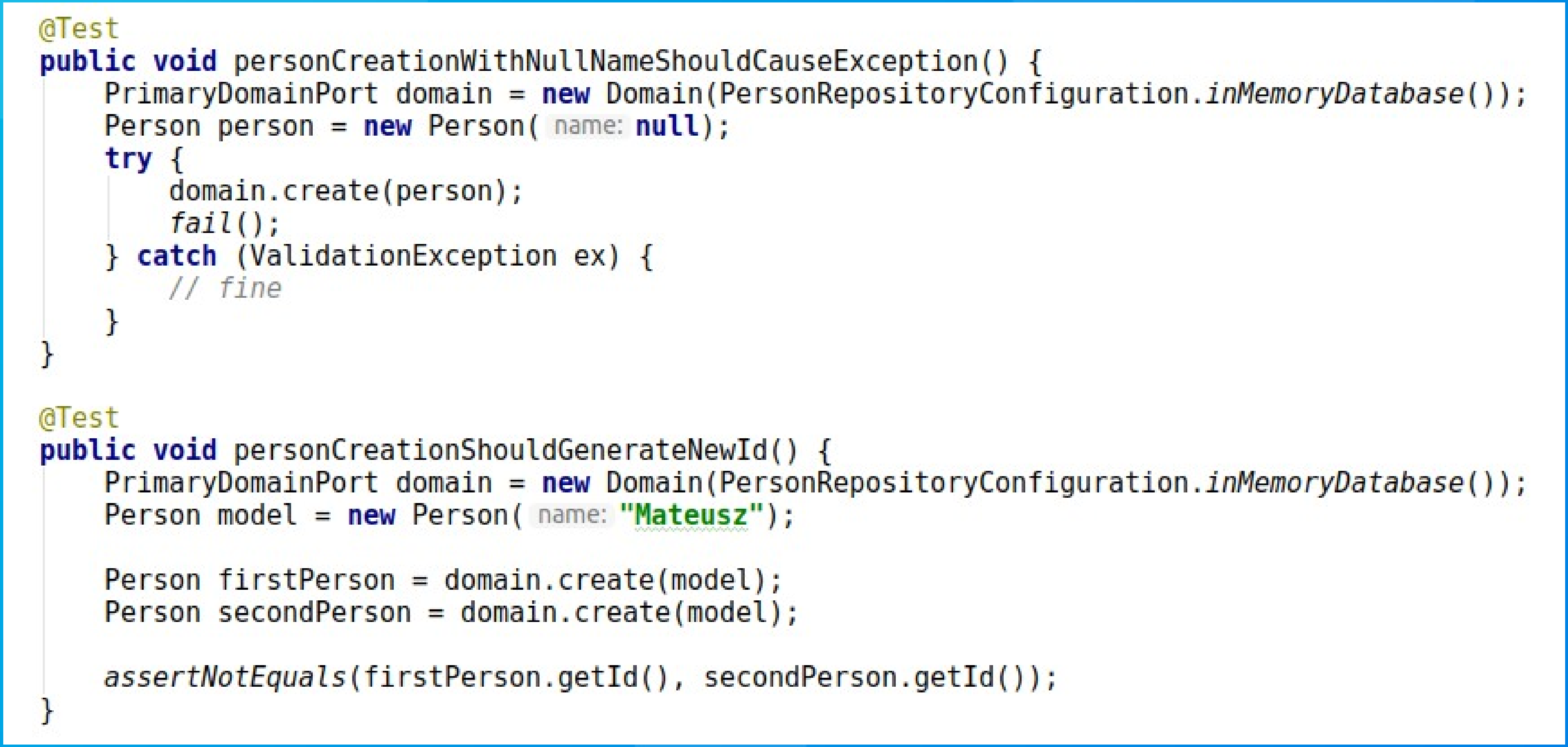
```java
public interface PersonRepository {

    Person create(Person person);

    void update(PersonId personId, Person person);

    Person findById(PersonId personId);

}
```

3. For each secondary port provide mock adapter



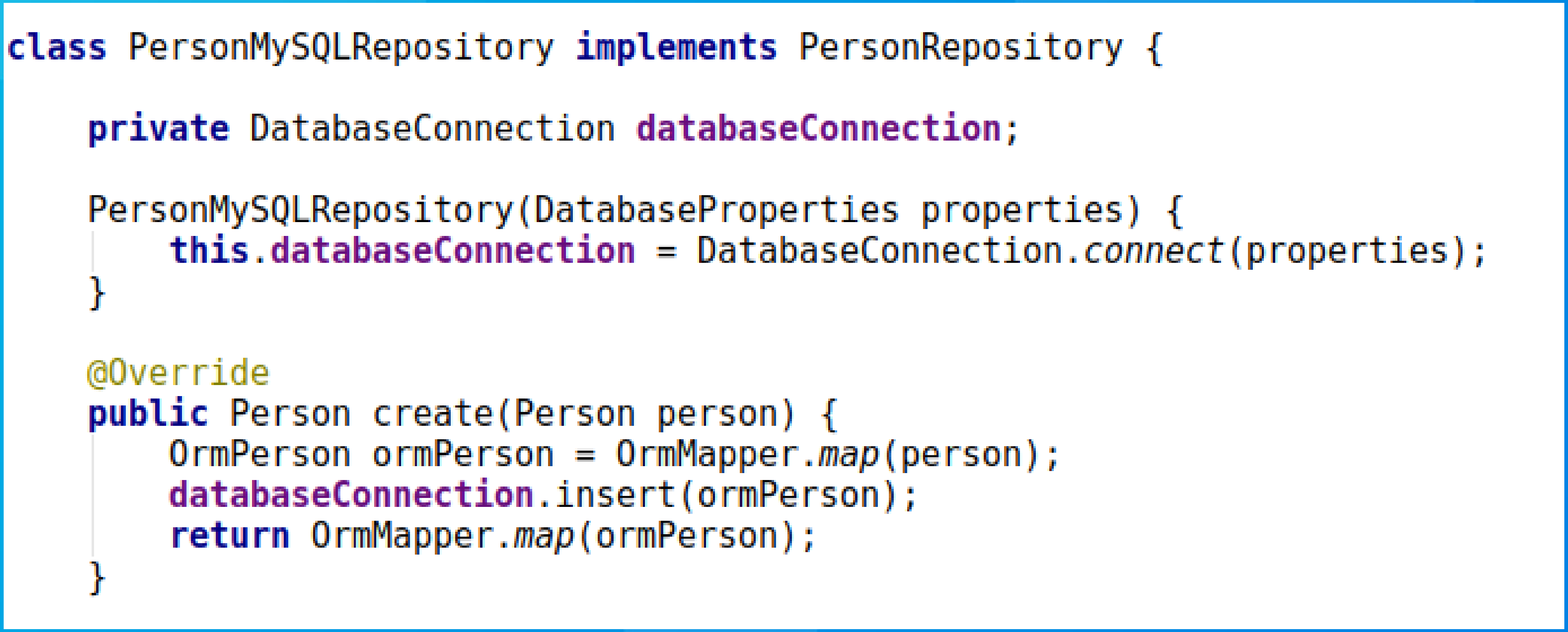
```java
class PersonInMemoryRepository implements PersonRepository {

    private final Map<PersonId, Person> database;

    PersonInMemoryRepository() {
        this.database = new HashMap<>();
    }

    @Override
    public Person create(Person person) {
        PersonId personId = PersonId.generate();
        database.put(personId, person);
        return Person.withId(personId, person);
    }

    @Override
    public void update(PersonId personId, Person person) {
        database.put(personId, person);
    }

    @Override
    public Person findById(PersonId personId) {
        return database.get(personId);
    }

}
```

4. Use BDD/TDD to create your domain implementation with help of your mocked adapters



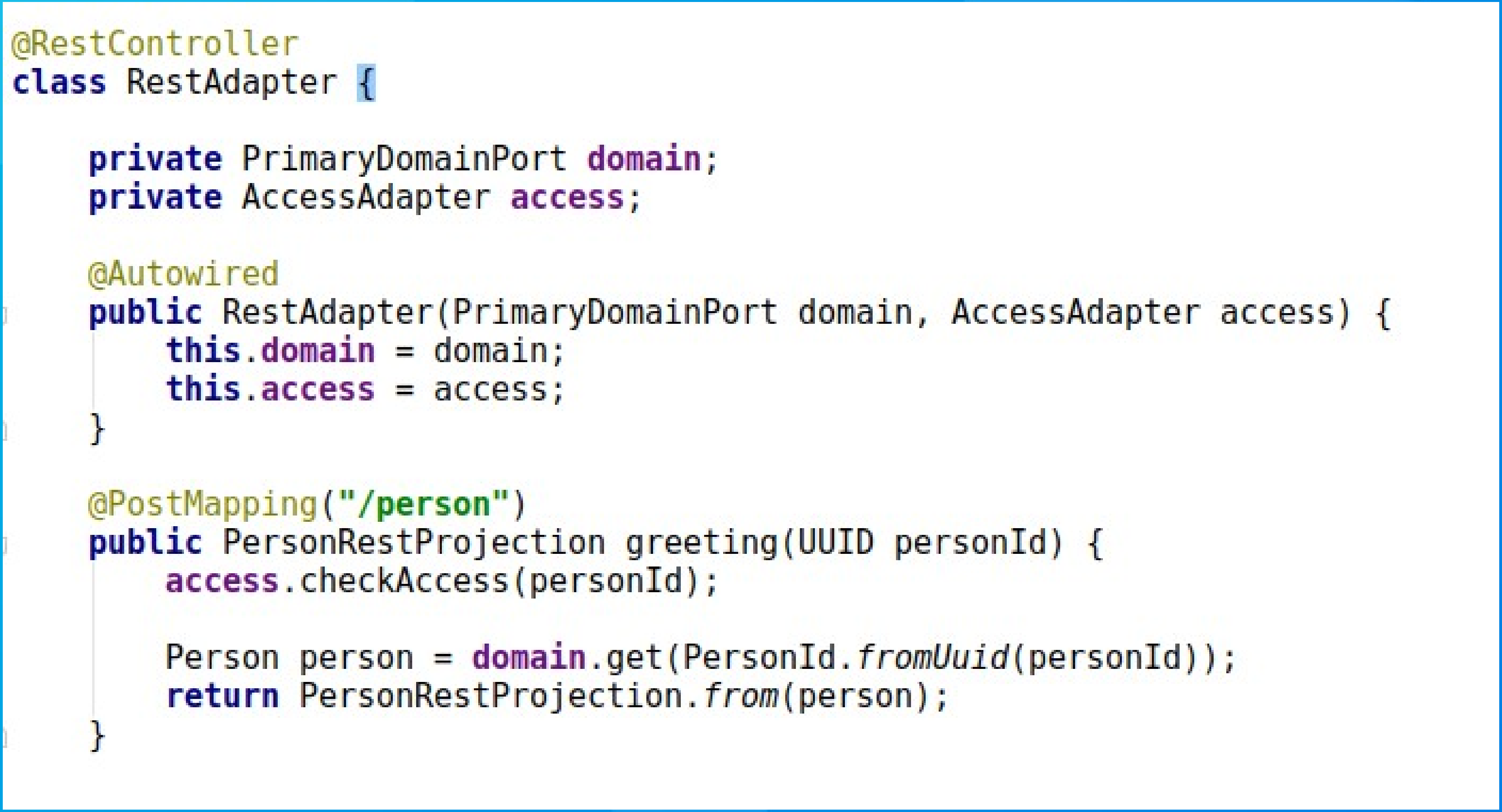
```
@Test
public void personCreationWithNullNameShouldCauseException() {
    PrimaryDomainPort domain = new Domain(PersonRepositoryConfiguration.inMemoryDatabase());
    Person person = new Person( name: null);
    try {
        domain.create(person);
        fail();
    } catch (ValidationException ex) {
        // fine
    }
}

@Test
public void personCreationShouldGenerateNewId() {
    PrimaryDomainPort domain = new Domain(PersonRepositoryConfiguration.inMemoryDatabase());
    Person model = new Person( name: "Mateusz");

    Person firstPerson = domain.create(model);
    Person secondPerson = domain.create(model);

    assertNotEquals(firstPerson.getId(), secondPerson.getId());
}
```

5. Create real secondary adapters (create some unit/integration tests for them)



```java
class PersonMySQLRepository implements PersonRepository {

    private DatabaseConnection databaseConnection;

    PersonMySQLRepository(DatabaseProperties properties) {
        this.databaseConnection = DatabaseConnection.connect(properties);
    }

    @Override
    public Person create(Person person) {
        OrmPerson ormPerson = OrmMapper.map(person);
        databaseConnection.insert(ormPerson);
        return OrmMapper.map(ormPerson);
    }
```

6. Create primary adapters



```java
@RestController
class RestAdapter {

    private PrimaryDomainPort domain;
    private AccessAdapter access;

    @Autowired
    public RestAdapter(PrimaryDomainPort domain, AccessAdapter access) {
        this.domain = domain;
        this.access = access;
    }

    @PostMapping("/person")
    public PersonRestProjection greeting(UUID personId) {
        access.checkAccess(personId);

        Person person = domain.get(PersonId.fromUuid(personId));
        return PersonRestProjection.from(person);
    }
```

Benefits of hexagonal architecture?

Testability!

Maintability/ Technical Debt

Flexibility of swapping technologies

Flexibility of swapping technologies

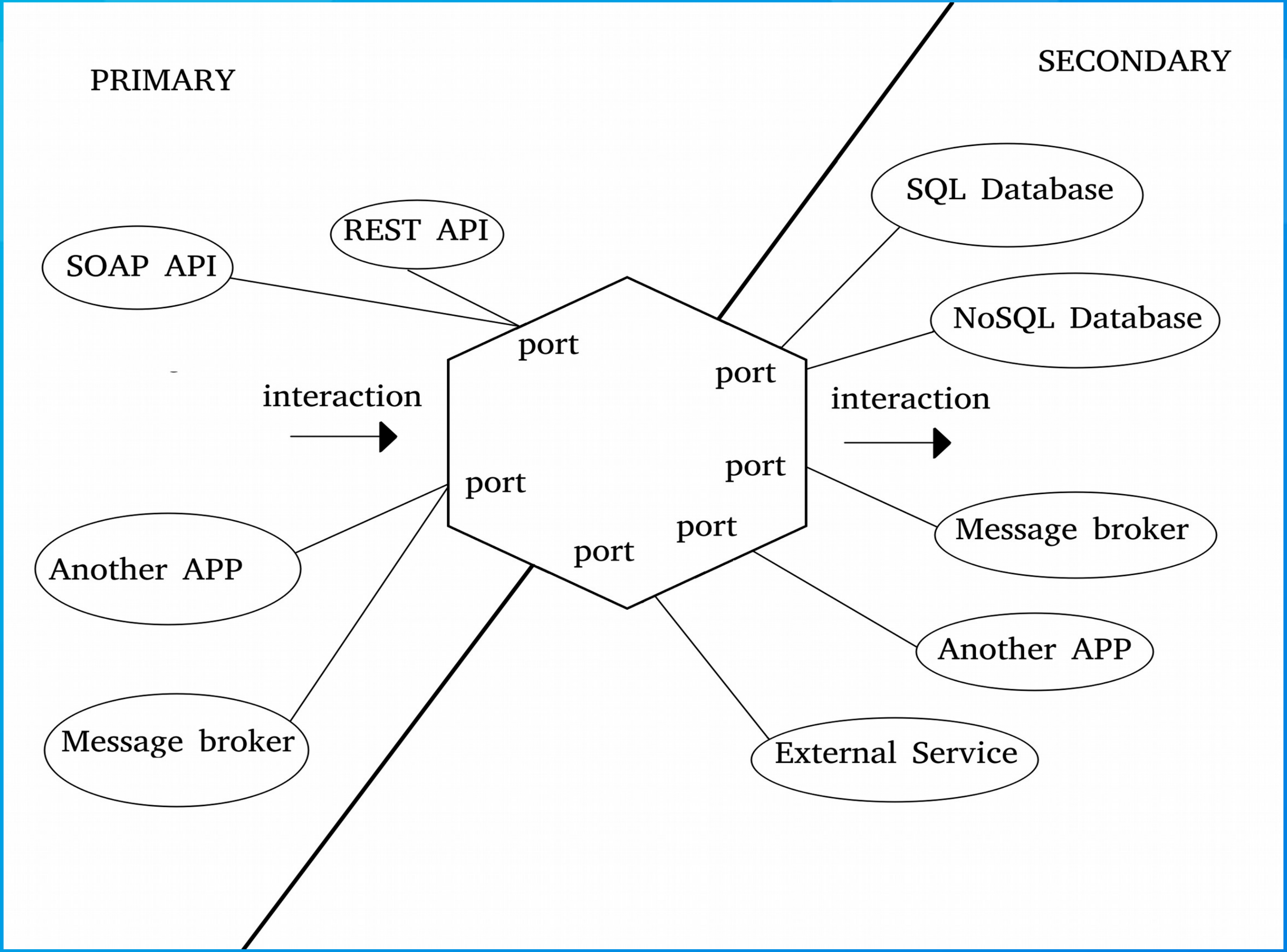
Downsides of hexagonal architecture?

Additional abstractions and complexity

Sometimes we lose framework power

When to go hexagonal?

Q&A



PRIMARY
SECONDARY
SOAP API
REST API
SQL Database
NoSQL Database
port
port
port
port
port
port
interaction
interaction
Another APP
Message broker
Message broker
Another APP
External Service