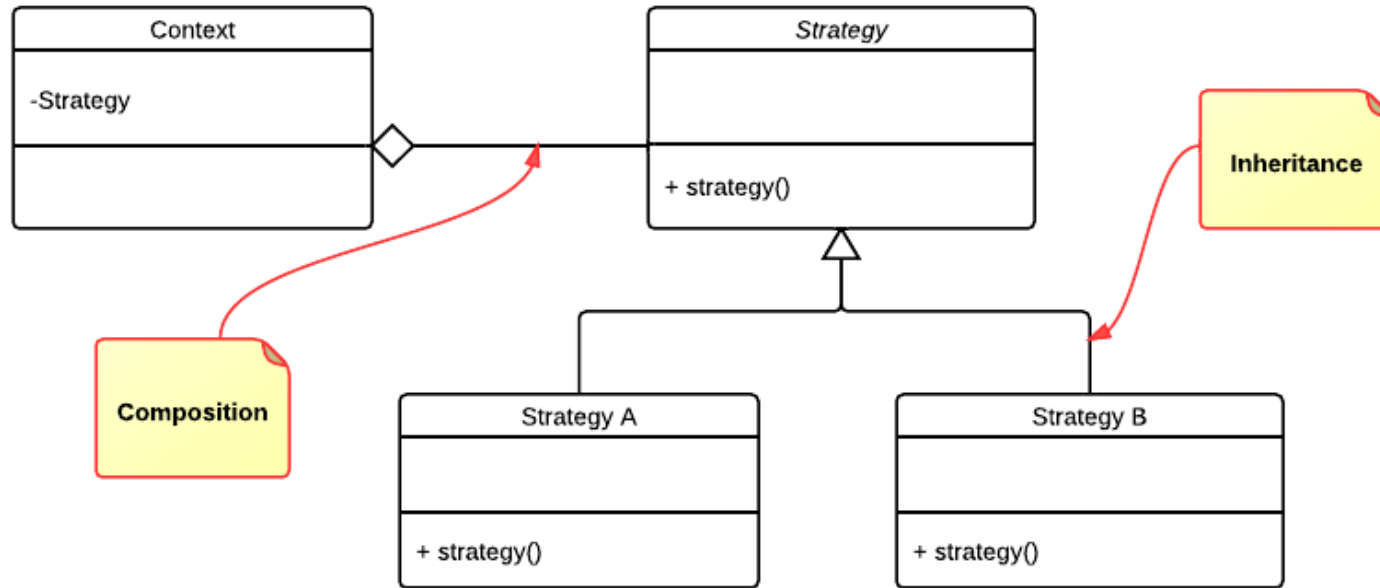


Relationships

In OO there are two ways to "connect" classes. **Composition** (almost always the right choice) and **Inheritance**



In the following we will see how these two strategies can be mapped to a DataBase, using JPA

All Possible Cardinality-Direction Combinations

Use the links below, for detailed information related to each type

Cardinality	Direction
One-to-one details	Unidirectional
One-to-one	Bidirectional
One-to-many details	Unidirectional
One-to-many	Bidirectional
Many-to-one details	Unidirectional
Many-to-one	Bidirectional
Many-to-many details	Unidirectional
Many-to-many	Bidirectional

We will investigate this in details in todays exercises

Bidirectional relationships

Rules that applies to bidirectional relationships:

The inverse side of a bidirectional relationship must refer to its owning side by use of the **mappedBy** element of the **OneToOne**, **OneToMany**, or **ManyToMany** annotation. The **mappedBy** element designates the property or field in the entity that is the owner of the relationship.

```
public class Customer .. {
```

```
...
```

```
    @OneToMany(mappedBy = "customer")  
    private List<Address> addresses = new ArrayList();
```

```
@Entity  
public class Address ..{  
    private static final long serialVersionUID = 1L;  
    ..  
    @ManyToOne  
    private Customer customer;
```



Side with the Foreign Key

Bidirectional relationships the *mappedBy* element

The **mappedBy** element designates the property or field in the entity that is the owner of the relationship.


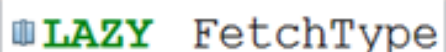
- The many side of one-to-many / many-to-one bidirectional relationships must be the owning side, hence the `mappedBy` element cannot be specified on the `ManyToOne` annotation.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships either side may be the owning side

Relationships - Lazy Fetching

The cost of retrieving and building an object's relationships far exceeds the cost of selecting the object

The solution to this issue is **lazy fetching** (lazy loading). Lazy fetching allows the fetching of a relationship to be deferred until it is accessed

Lazy fetching involves some *magic* in the JPA provider to transparently fault in the relationships as they are accessed.

```
@OneToOne(fetch = FetchType.  
@JoinColumn(name="ADDR_ID")   
private Address address;
```

Relationships - Cascading

Relationship mappings have a cascade option that allows the relationship to be cascaded for common operations.

Cascade is normally used to model dependent relationships, such as Order -> OrderLine.

Cascading the orderLines relationship allows for the Order's -> OrderLines to be persisted, removed, merged along with their parent.

```
@OneToOne(cascade={CascadeType.  
@JoinColumn(name="ADDR_ID")  
private Address address;
```

ALL	CascadeType
DETACH	CascadeType
MERGE	CascadeType
PERSIST	CascadeType
REFRESH	CascadeType
REMOVE	CascadeType