Object-Relational Mapping

- Architecture of an ORM Framework
- Pattern for Object-Relational Mapping
- ORM - Technology Case Study: JPA
- Similar concept: Data Binding

Mapping Object Oriented to Relational Concepts

Example:

```
public class Person {
    private String name;
    private List<Car> cars;
    …
}

public class Car {
    private String model;
    …
}

…

Person p1=new Person("Alice");
p.addCar(new Car("BMW"));
p.addCar(new Car("Ferrari"));
```

Table Person

ID	name
01	Alice
02	Bob

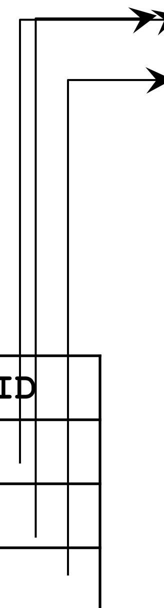


Table Car

model	PID
BMW	01
Ferrari	01
Toyota	02

Example with JDBC

- Saving objects to tables example: Ad-hoc manual mapping with JDBC:
- JDBCPersonsCars.zip

The “Impedance Mismatch” Problem

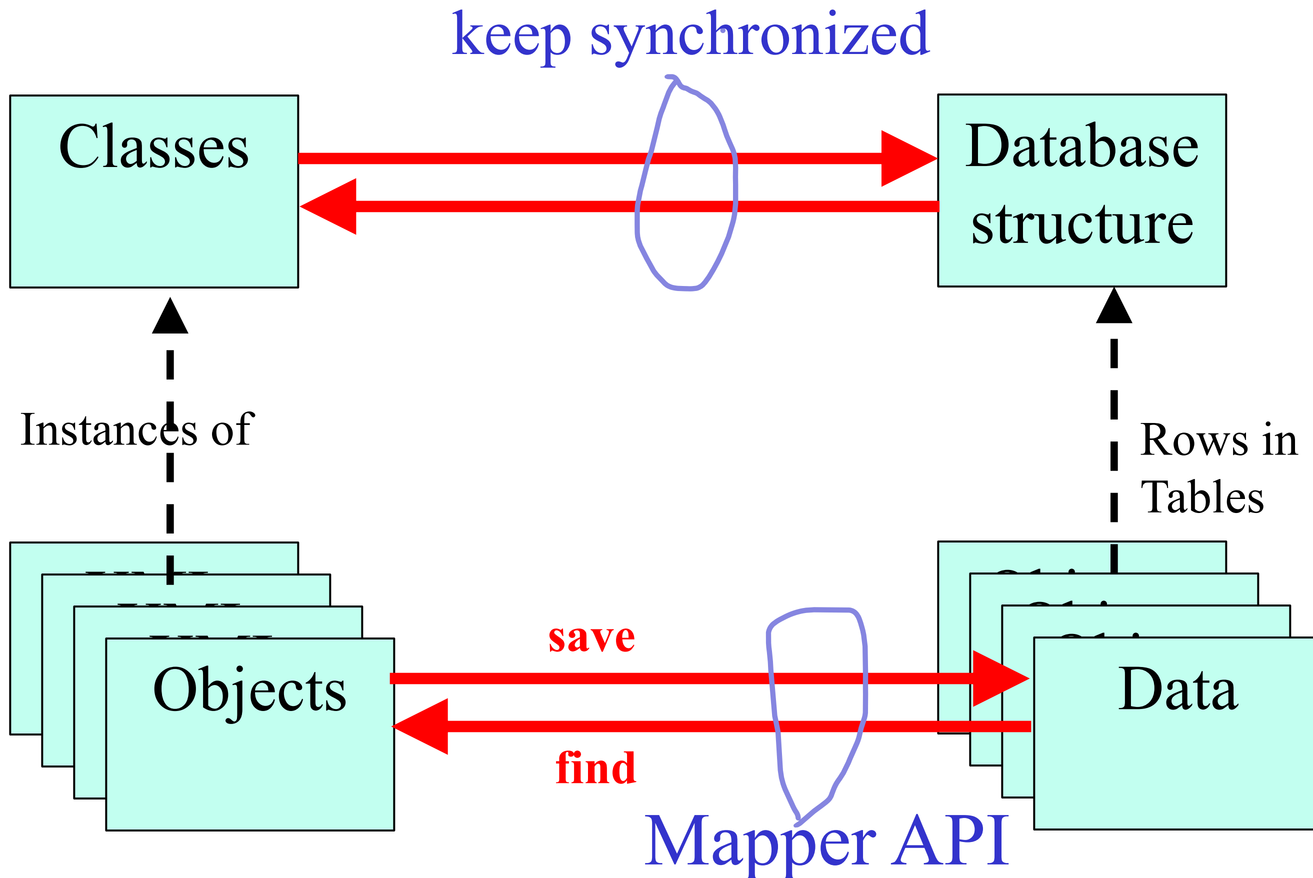
- **Problem: Data source (relational) and OO program use different data modeling concepts**
  - ➢ **this *conceptual gap* is called “*Impedance mismatch*”**

- Concepts: Elements and Relationships:
  - OO programs:
    - classes, objects, attributes;
    - relations: association, aggregation, inheritance
  - Relational databases:
    - tables, rows, columns;
    - relations: foreign key references

The “Impedance Mismatch” Problem

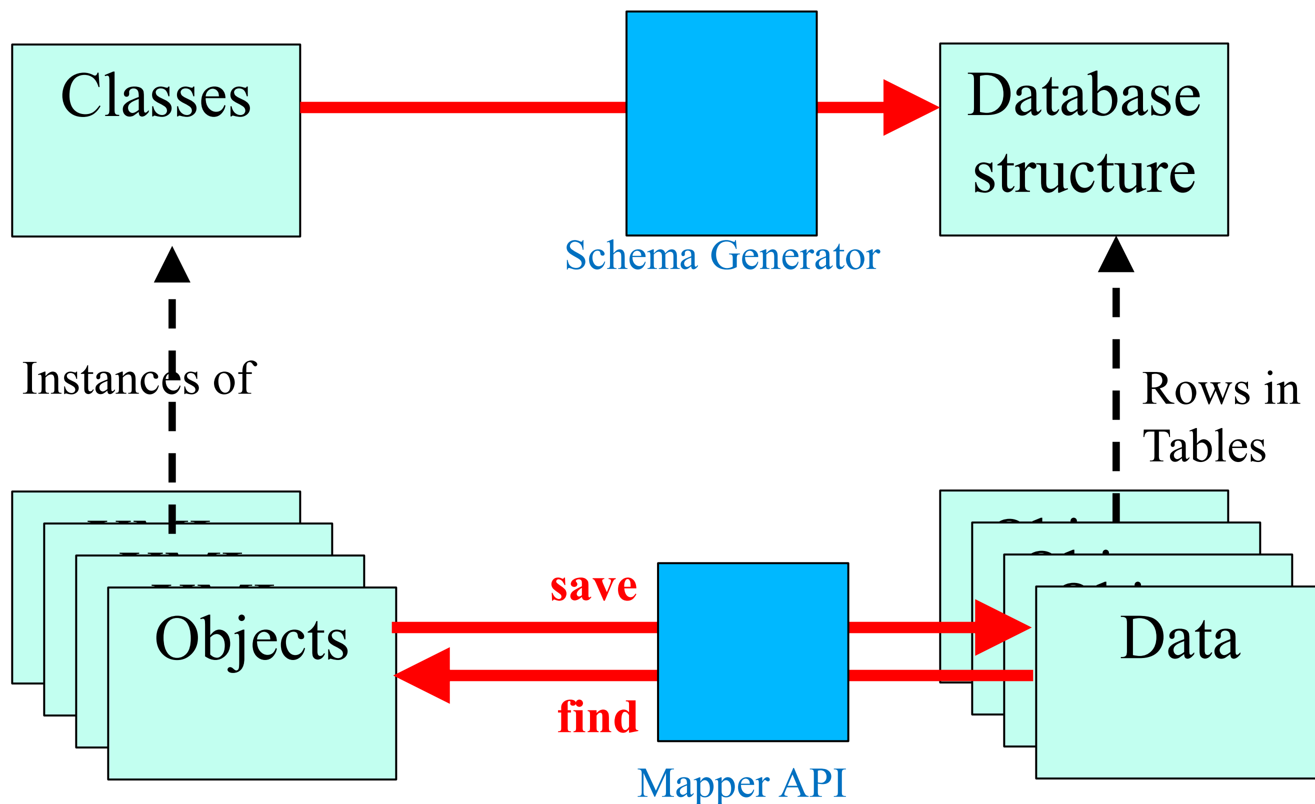
- Solution: Mapping OO concepts to Relational concepts:
  - Mapping patterns: Class diagram <-> Database schema
    - Mapping pattern establish **rules** on how to map classes and types of relationship that can appear in a class diagram to tables and relationships in a database schema
  - Automatic mapping tools and frameworks:
    - Object-Relational Mapper (ORM):
      - an abstraction layer that allows application developers to interact with databases using objects instead of writing raw SQL queries.
      - it **reduces boilerplate code**
    - ORM examples:
      - Java: JPA, Hibernate, EclipseLink
      - .NET: Entity Framework
      - Python: Django ORM, SQLAlchemy

Object Relational Mapping

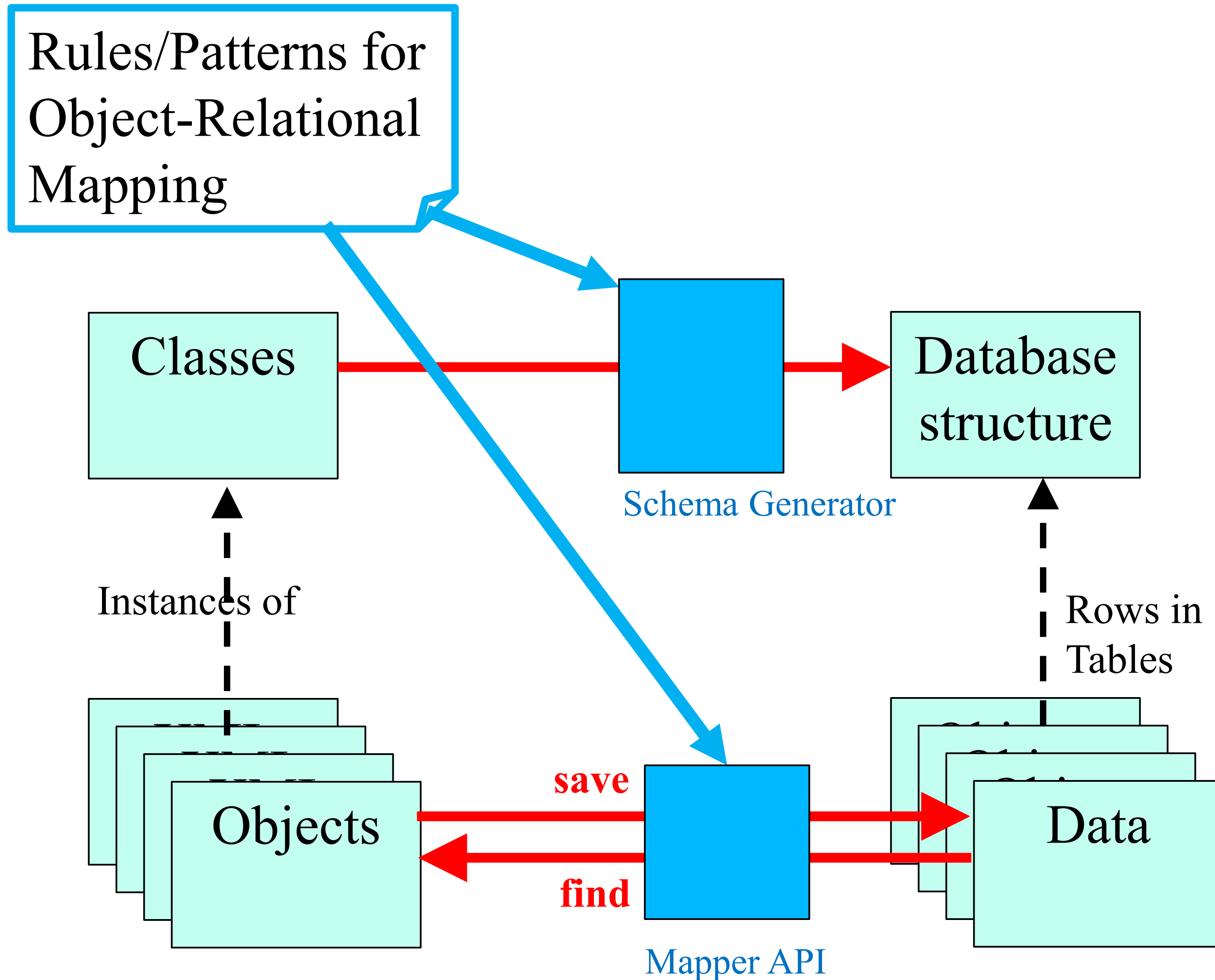


Automatic ORM Tools/Frameworks

ORM = Generator + Mapper



Automatic ORM Tools/Frameworks



Mapping Object Oriented to Relational Concepts

Simple Example:

```
public class Person {
     private String name;
     private int age;
     Person(String n, int a) {
         name=n;
         age=a;
     }
}

…

Person p1=new Person("Alice", 21);
Person p2=new Person("Bob", 19);
```

Table Person

name	**age**
Alice	21
Bob	19

Mapping Object Oriented to Relational Concepts

- General rules:
  - Class <-> Table
  - Class Attribute <->Column in Table
  - Object (instance of Class)<-> Row in Table
  - Works well only for simple classes with scalar attributes only
- Problems:
  - How to identify objects? Is there an attribute usable as primary key?
  - Mapping of class relationships:
    - Aggregations
    - Associations
    - Inheritance

Mapping Object Oriented to Relational Concepts

Example:

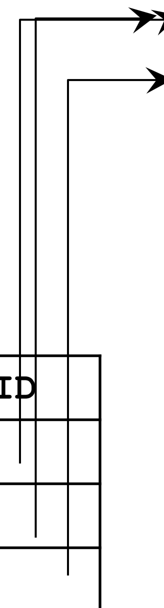
```
public class Person {
    private String name;
    private List<Car> cars;
    …
}

public class Car {
    private String model;
    …
}

…

Person p1=new Person("Alice");
p.addCar(new Car("BMW"));
p.addCar(new Car("Ferrari"));
```

Table Person

ID	name
01	Alice
02	Bob

Table Car

model	PID
BMW	01
Ferrari	01
Toyota	02



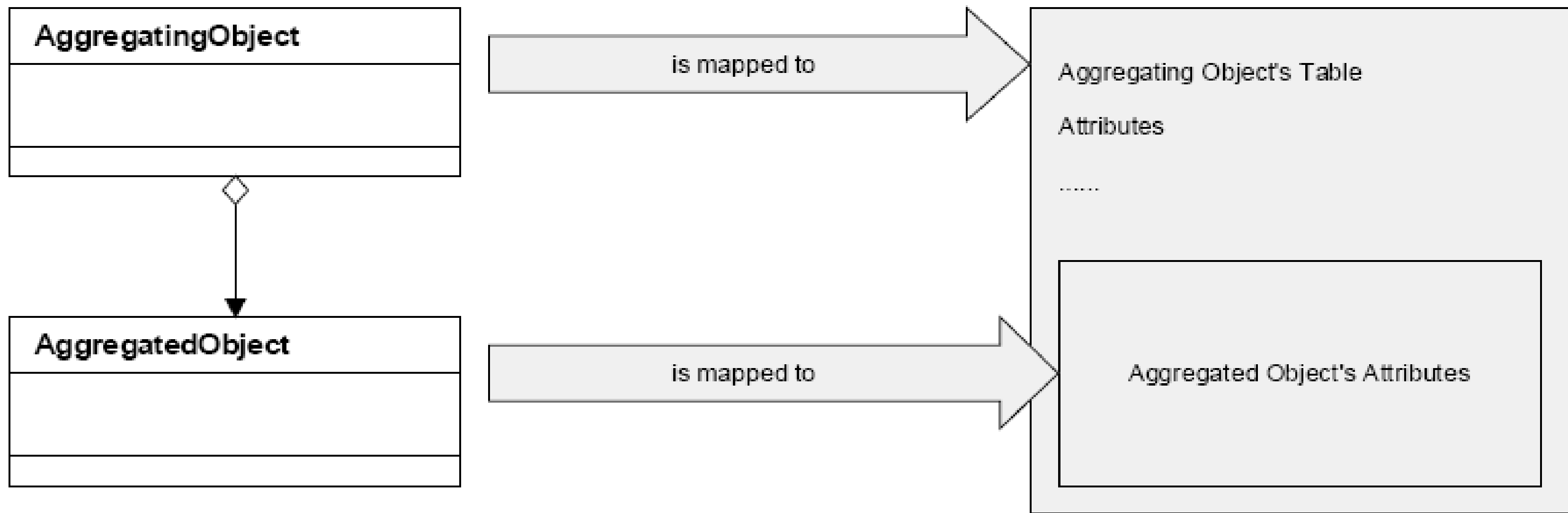
Pattern for Object-Relational Mapping

Pattern for OO-Relational Mapping

- How to map each type of class diagram relationship into database schema relationships?
  - Aggregation
  - Association
  - Inheritance

- Wolfgang Keller, *Mapping Objects to Tables* article

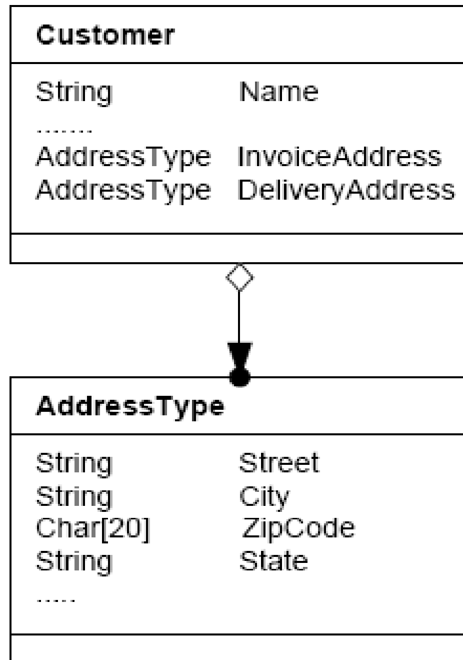
Solutions for Mapping Aggregation (1): Single Table Aggregation



Solution:
Put the aggregated object's attributes into the same table as the aggregating object’s.

Figure from : Wolfgang Keller, Mapping Objects to Tables

Single Table Aggregation Example & Consequences



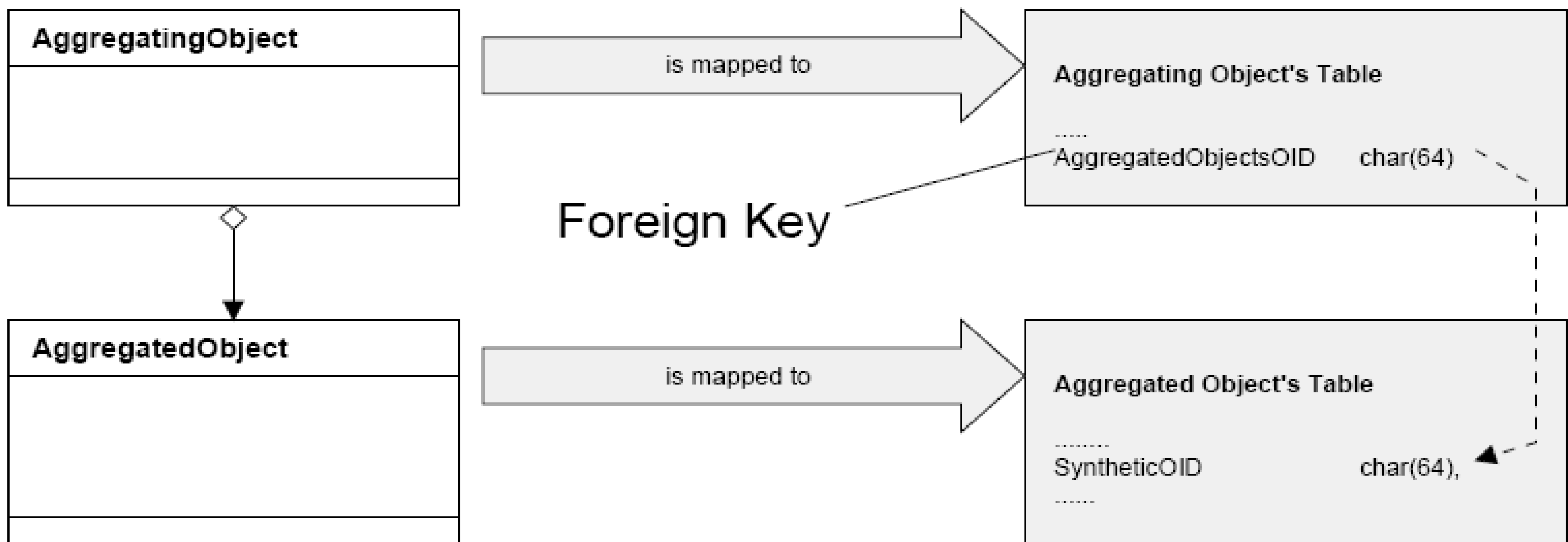
CustomerTable	
Name	char(50)
......	
InvAdrStreet	char(50)
InvAdrCity	char(50)
InvAdrZipCode	char(20)
InvAdrState	char(50)
DelAdrStreet	char(50)
DelAdrCity	char(50)
DelAdrZipCode	char(20)
DelAdrState	char(50)
....	

Performance: **+**, only one table needs to be accessed for queries

Flexibility: **-**, if the aggregated object type is aggregated in more than one object type, the design results in poor maintainability

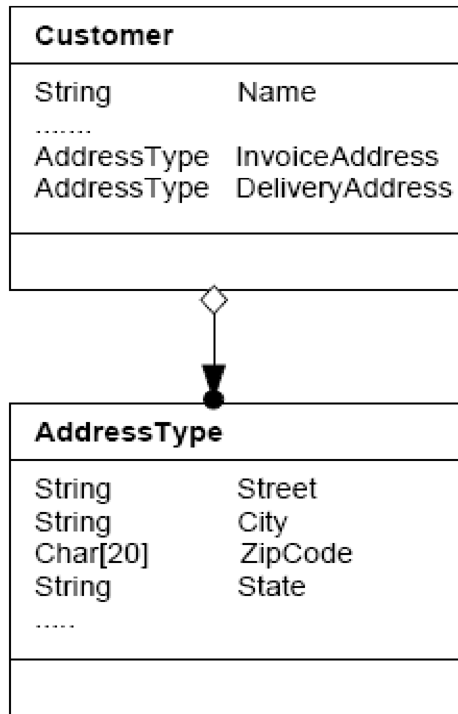
Figure from : Wolfgang Keller, Mapping Objects to Tables

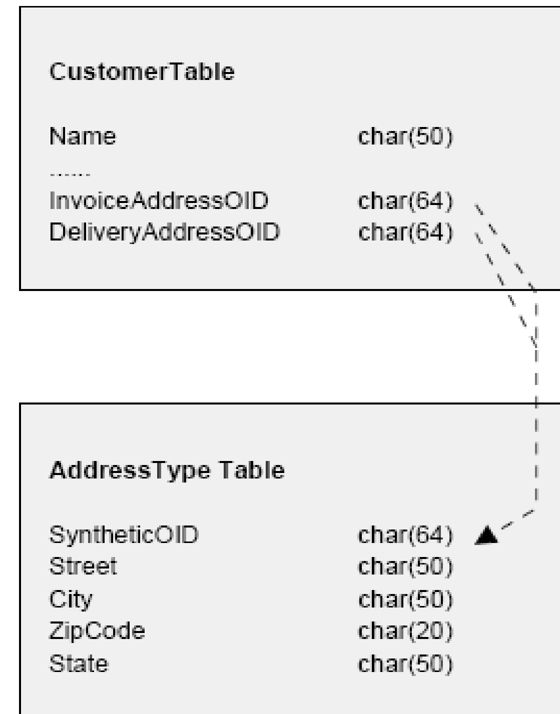
Solutions for Mapping Aggregation (2): Foreign Key Aggregation



Solution:
Use a separate table for the aggregated type. Insert an Synthetic Object Identity into the table and use this object identity in the table of the aggregating object to make a foreign key link to the aggregated object.

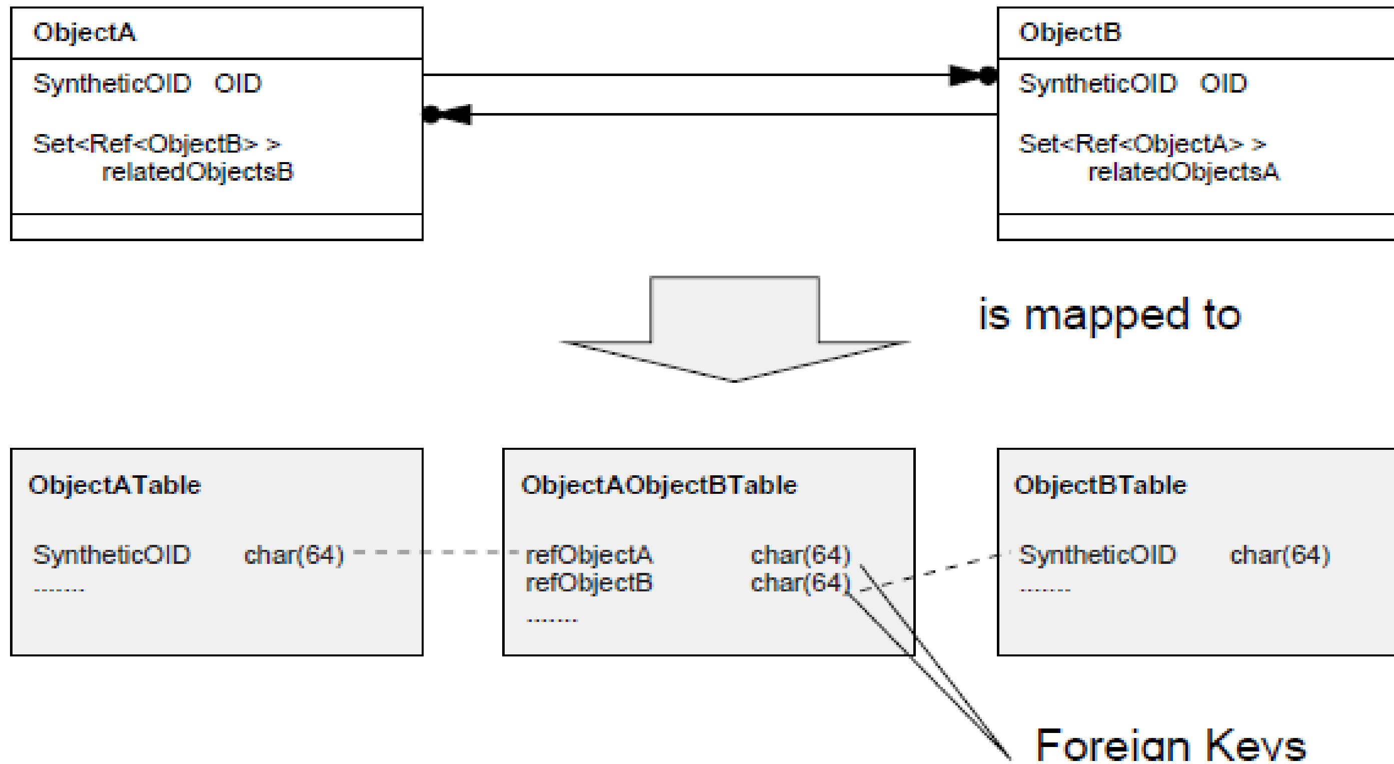
Foreign Key Aggregation Example & Consequences





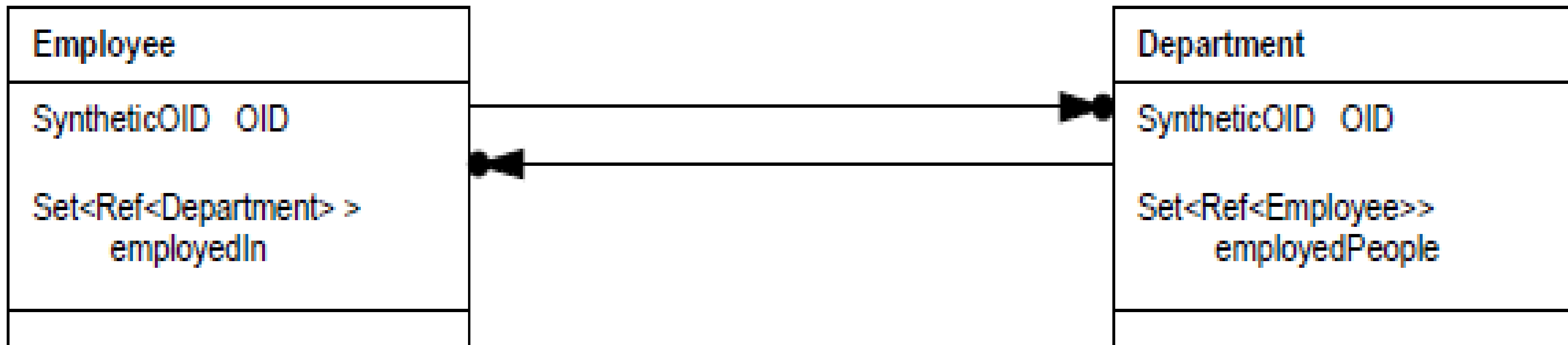
Performance: -, needs a join operation or at least two database accesses
Maintenance: +, Factoring out objects into tables of their own makes them easier to maintain and hence makes the mapping more flexible.
Consistency of the database: !, Aggregated objects are not automatically deleted on deletion of the aggregating objects.

Solution for Mapping Associations: Association Table



Solution: Create a separate table containing the Object Identifiers (or Foreign Keys) of the two object types participating in the association. Map the rest of the two object types to tables using any other suitable mapping patterns presented in this paper.

Association Table Example

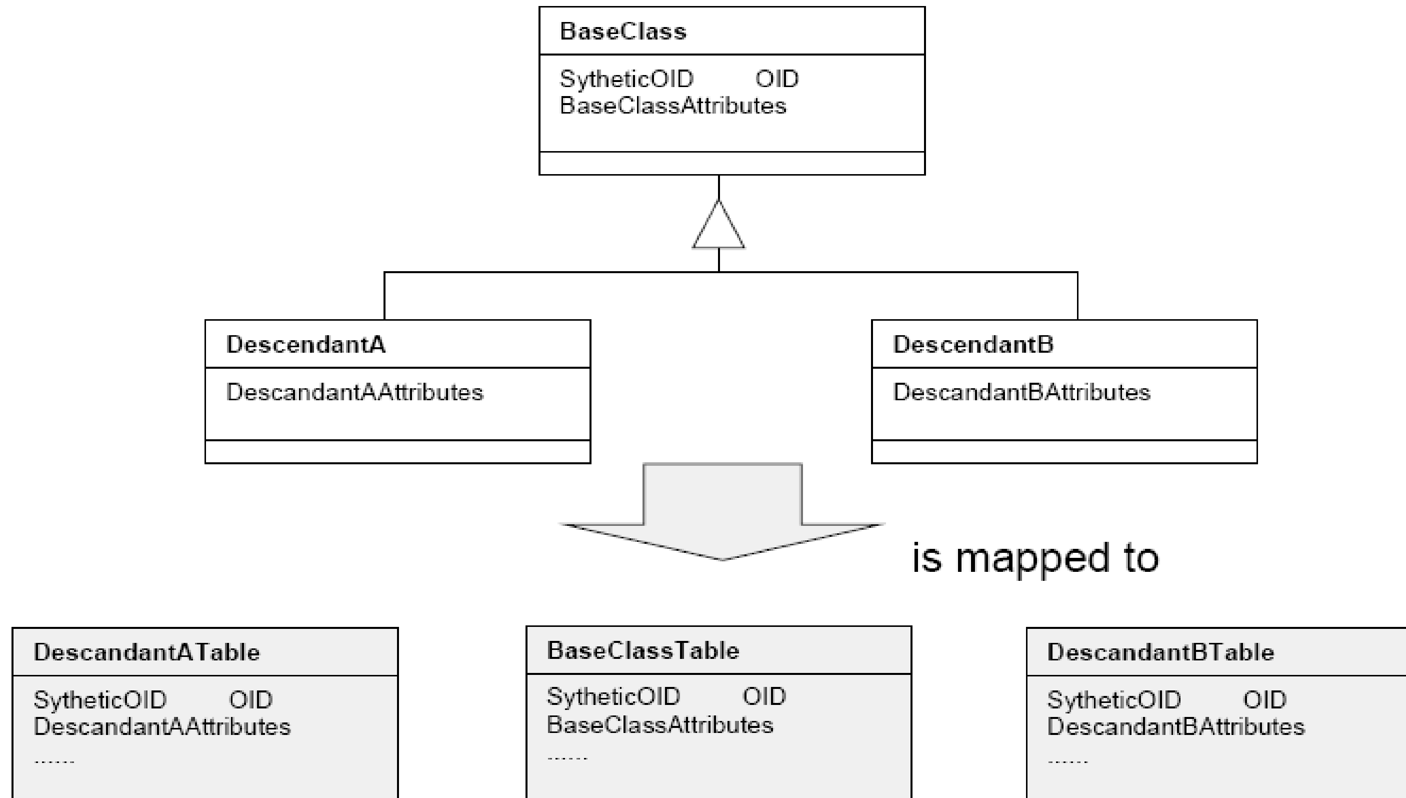


n:m association between Employees and Departments

An Employee may work with m Departments

A Department comprises n Employees

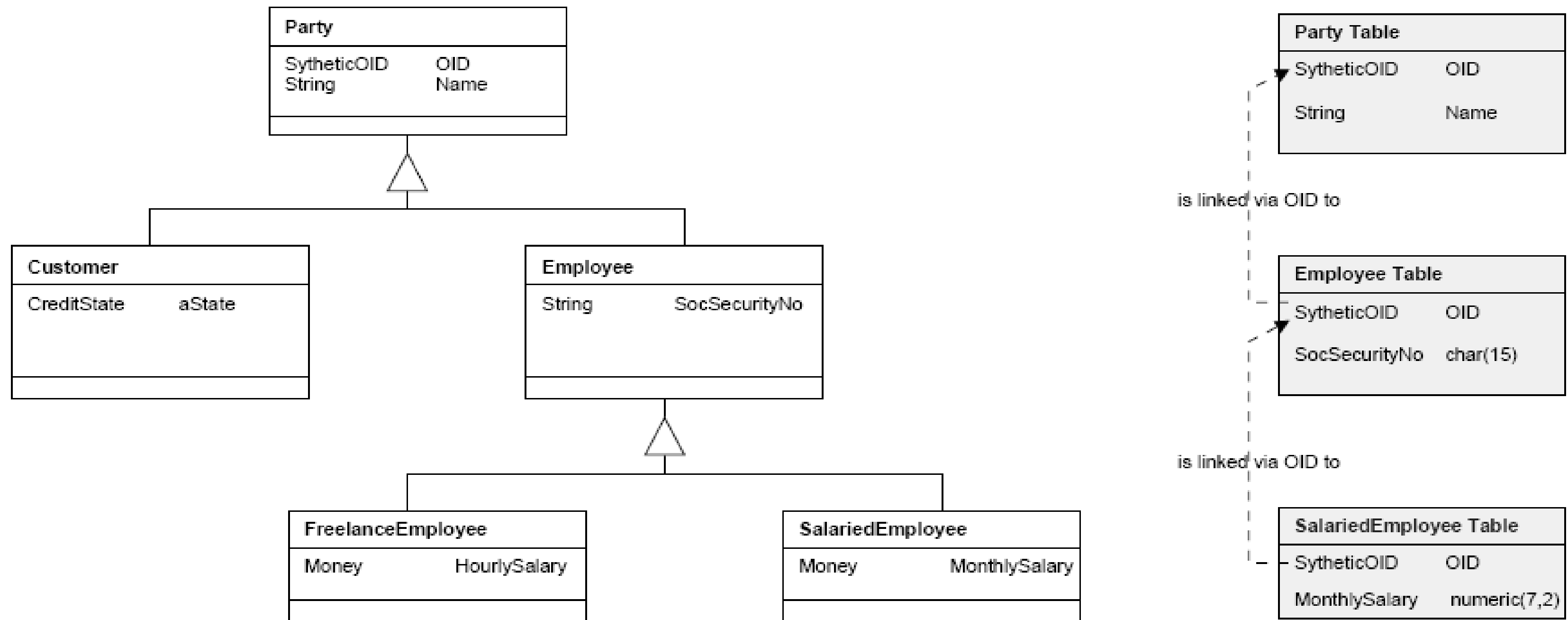
Solutions for Mapping Inheritance (1): One Class – One Table



Solution:

Map the attributes of each class to a separate table. Insert a Synthetic OID into each table to link derived classes rows with their parent table's corresponding rows.

One Class – One Table Example & Consequences



Performance: -, reading a SalariedEmployee instance in our example costs 3 (=depth of inheritance tree) database read operations

Solutions for Mapping Inheritance (2): One Inheritance Tree – One Table

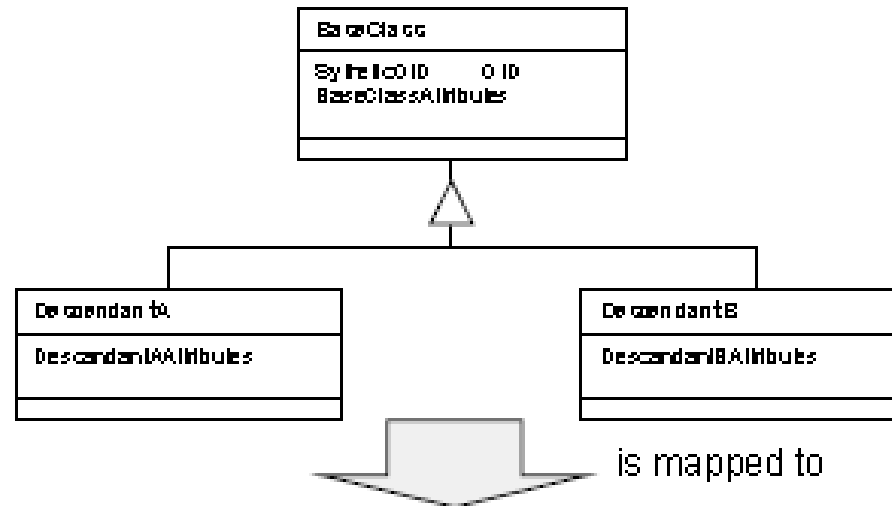


Table for BaseClass, DescendantA, DescendantB

	SyntheticOID, BaseClassAttributes	DescendantAAttributes	DescendantBAttributes
BaseClass instance	Attribute Values	Null Values	Null Values
DescendantA instance	Attribute Values	Attribute Values	Null Values
DescendantB instance	Attribute Values	Null Values	Attribute Values

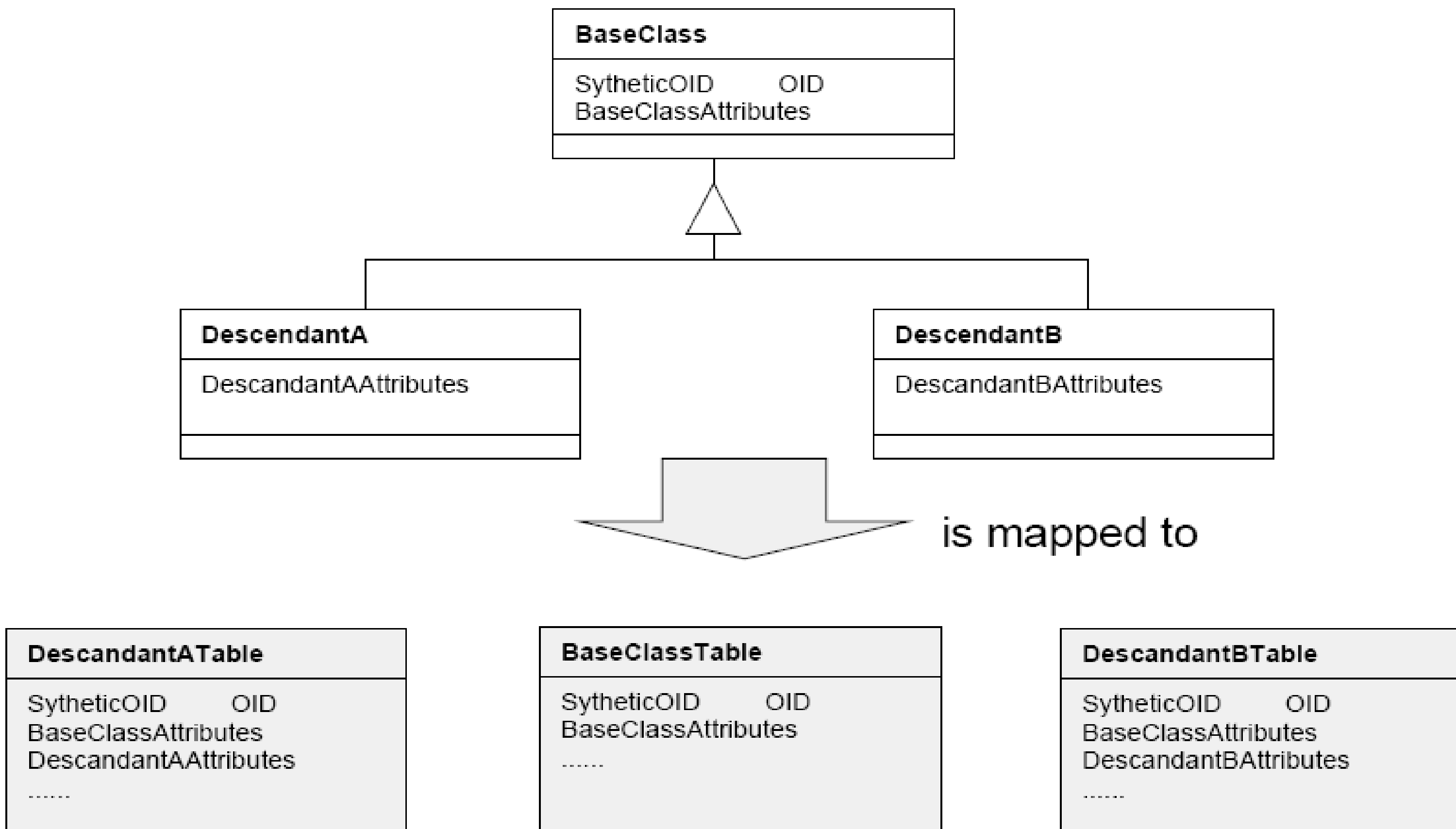
Use the union of all attributes of all objects in the inheritance hierarchy as the columns of a single database table. Use Null values to fill the unused fields in each record.

One Inheritance Tree – One Table Example & Consequences

```
Table PartyHierarchy

// Party attributes
SyntheticOID                  char(64)
Name                          char(50)
....
//Customer attributes
aState                        char(5)
....
// Employee attributes
SocSecurityNo                 char(15)
....
// FreelanceEmployee attributes
HourlySalary                  numeric(5,2)
....
//SalariedEmployee attributes
MonthlySalary                 numeric(7,2)
....
```

Performance: +, needs one database operation to read or write an object.
Space consumption: -, optimal space consumption.
Balancing database load across tables: -

Solutions for Mapping Inheritance (3): One Inheritance Path – One Table



Solution: Map the attributes of each concrete class to a separate table. To a classes' table add the attributes of all classes the class inherits from.

One Inheritance Path – One Table Example & Consequences

```
Table SalariedEmployee

// Party attributes
SyntheticOID                char(64)
Name                        char(50)
....

// Employee attributes
SocSecurityNo               char(15)
....

// SalariedEmployee attributes
MonthlySalary               numeric(7,2)
....
```

Performance: +, needs one database operation to read or write an object.
Space consumption: +, optimal space consumption.
Maintenance cost: +/-: inserting a new subclass means updating all polymorphic search queries. The structure of the tables remains untouched. Adding or deleting attributes of a superclass results in changes to the tables of all derived classes.

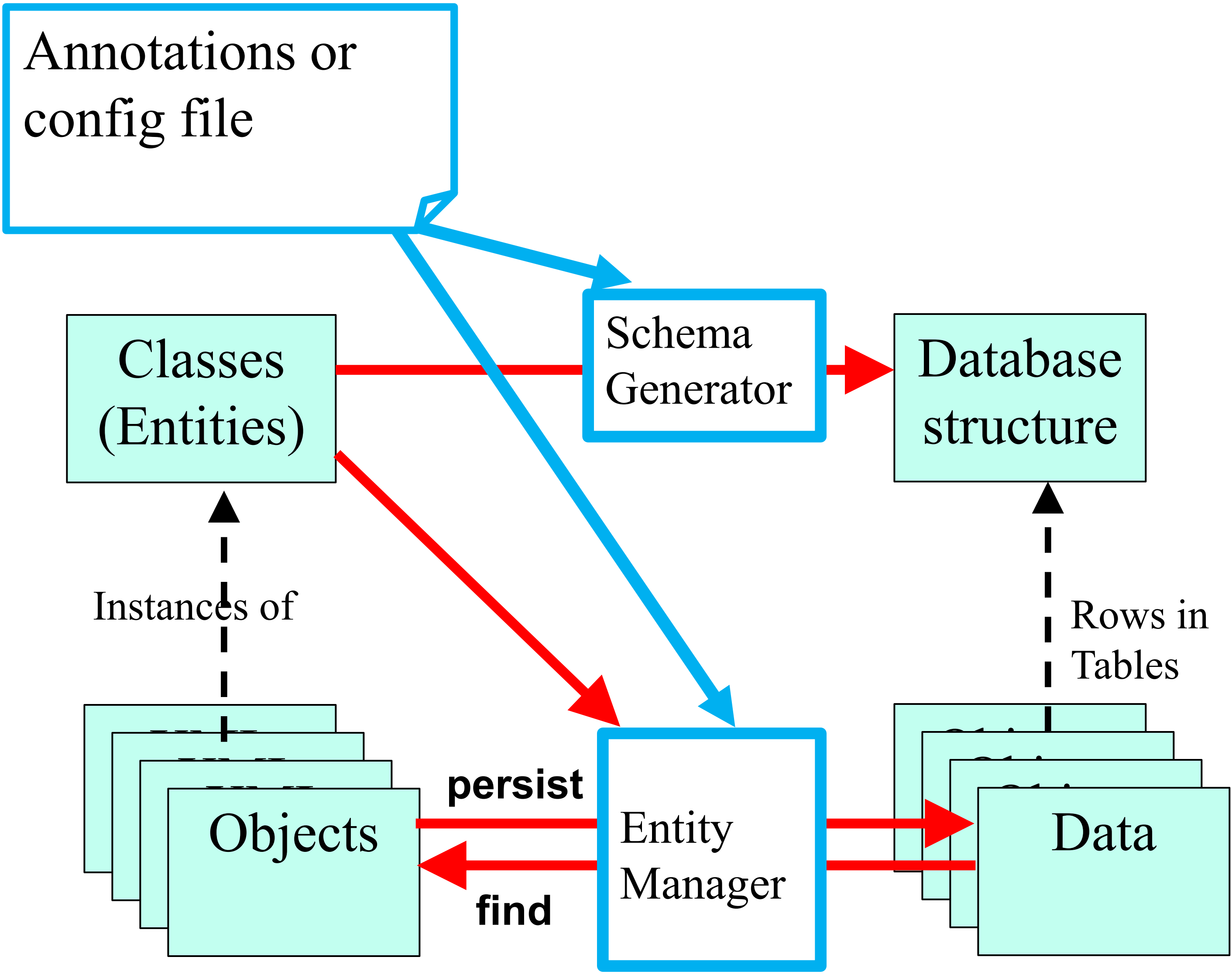
ORM Technologies- Case Study - JPA

Example: Java Persistence API (JPA)

- JPA = Java Persistence API
- JPA = Jakarta Persistence API
- Since 2020: javax.persistence.* => jakarta.persistence.*
- Defines an ***API*** for ORM
  - Has ***multiple implementations*** (Hibernate, EclipseLink, Apache OpenJPA)

- Fundamental concept: **Entities**
  - **Entity = a Class that can be processed by the ORM**
  - Has to be explicitly be marked as entity (annotations or config file)
- EntityManager

JPA



Simple Example with JPA

```
@Entity
public class Employee {
// …
}
```

```
// ...
EntityManagerFactory emfactory =
    Persistence.createEntityManagerFactory( “demoPU" );

EntityManager emanager = emfactory.createEntityManager( );

entitymanager.getTransaction( ).begin( );

Employee employee = new Employee(“John”, “Doe”, 12345);

emanager.persist( employee );

emanager.getTransaction( ).commit( );

emanager.close( );

emfactory.close( );
```

API vs Implementation

- JPA is an API (Application Program Interface)
  - There are several implementations of the JPA API: Hibernate, EclipseLink
- Many API's (including JPA) use Factory Pattern

```
EntityManagerFactory emfactory =
     Persistence.createEntityManagerFactory( “demoPU" );

EntityManager emanager = emfactory.createEntityManager( );
```

The abstract factory takes as argument the name of a **persistence unit** defined in **persistence.xml** file and will use the configuration defined there

```
<persistence-unit name="demoPU">
    <properties>
      <!-- DB config -->
      <property name="jakarta.persistence.jdbc.url" value="jdbc:h2:mem:testdb"/>
      <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="jakarta.persistence.jdbc.user" value="sa"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- Hibernate -->
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
```

Simple Example with JPA

```
@Entity
public class Employee {
// …
}
```

```
// ...
EntityManagerFactory emfac
     Persistence.createEnt

EntityManager emanager = emfactory.

entitymanager.getTransaction( ).begin( );

Employee employee = new Employee("John", "Doe", 12345);

emanager.persist( employee );

emanager.getTransaction( ).commit( );

emanager.close( );

emfactory.close( );
```

For a class annotated as @Entity, JPA/Hibernate treats it as a persistent entity and can create a corresponding database table for it.

Schema-generation must be enabled in persistence.xml

```
<property name="hibernate.hbm2ddl.auto" value="create"/>
```

Choosing Mapping Pattern

- Can choose the mapping pattern (by annotations or config files)
- Inheritance Mapping Strategies: can be configured to one of:
  - SINGLE_TABLE (one inheritance tree - one table)
  - JOINED (one class – one table)
  - TABLE_PER_CLASS (table per concrete class : one inheritance path - one table)
- Aggregations:
  - Embeddable classes in Entities: single table aggregation
- Associations:
  - Characterized by Multiplicity and Direction: Foreign key or Association table
    - One-To-One
    - One-To-Many
    - Many-To-Many

Persons-Cars Example with JPA

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    @OneToMany(
            mappedBy = "person",
            cascade = CascadeType.ALL,
            orphanRemoval = true
    )
    private List<Car> cars = new ArrayList;

    public Person() {
    }

 …

}
```

Id is primary key. Database auto-increments ID

One Person can own many Car objects. Attribute mappedBy means the relationship is managed by the person field inside Car. CascadeType.ALL means if a person is saved so are automatically its cars.

Persons-Cars Example with JPA

```
@Entity
public class Car {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String model;

    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;

    public Car() {
    }

    …
}
```

Id is primary key. Database auto-increments ID

Many cars can belong to one person. The car table will contain a foreign key column

Similar Concept: Data Binding

- **Data binding** in the context of structured data (**XML** and **JSON)** means linking structured data to an application model so it can be automatically mapped, read, and saved
- Protobuf to object mapping is also a form of data binding: the protoc compiler generates data classes from proto files. Messages in the binary structured format of protobuf are serialized/deserialized between the binary structured format and data objects

- Similar to ORM: they all do **mapping between data representations and objects**

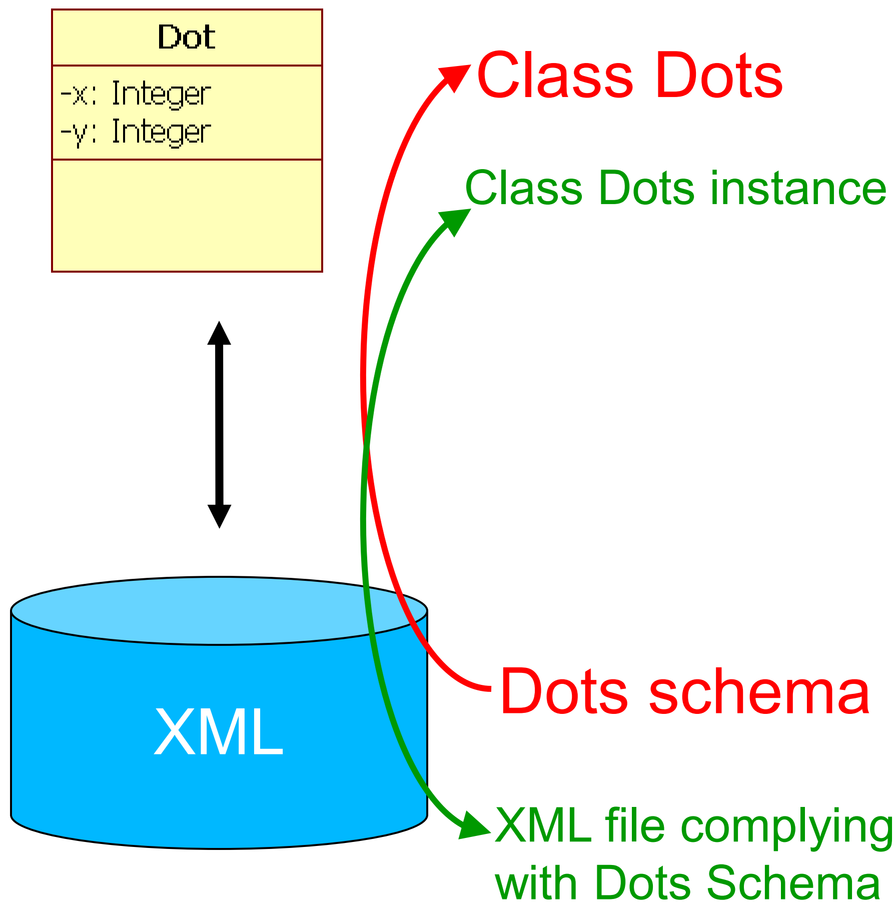
Example: XML to Object

dots.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<dots xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="dots.xsd">>
  <dot x="32" y="100" />
  <dot x="17" y="14" />
  <dot x="18" y="58" > </dot>
</dots>
```

XML files of the dot type should be loaded in memory as a list of elements of type Dot, where a Dot has 2 attributes (x and y)

Example: XML data binding



- **XML data binding: refers to a means of representing information from a XML document as a business object in memory**

- **<u>Tool-s can automatize the process of XML data binding</u>**: they create mappings between elements in a XML schema and the fields of a class

- Example: JAXB

Rules for Schema to Class mapping – Example

dots.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="dots">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="dot" maxOccurs="unbounded">
        <xs:complexType>
      <xs:attribute name="x" type="xs:integer" use="required"/>
      <xs:attribute name="y" type="xs:integer" use="required"/>
      </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

xs:elements of xs:complexType will be mapped as new generated classes

XML Data Binding