Data-driven systems

Design patterns in the data-access layer
Concurrency handling
Object-relational mapping

Automatizálási és
Alkalmazott
Informatikai Tanszék

Role of the data access layer

- Provide a high-level abstraction for data access
- Basic operations for data manipulation
- Concurrency handling

Object-relational mapping

Problem statement

Modelling

- Business Layer
  - Object oriented modelling
  - UML
  - Design Patterns
  - Not only data, but processes too
- Data Layer
  - Entity-Relationship diagram
  - UML data modelling profile
  - Static, attribute-based

Purpose of ORM

- Object Relational Mapping
  - Mapping business entities to the relational data model
  - Connect data storage and business workflows
- Problems
  - Different concepts
  - Inheritance
  - Shadow information
  - Relations

Sources used

- Examples are from:
  - http://www.agiledata.org/essays/mappingObjects.html

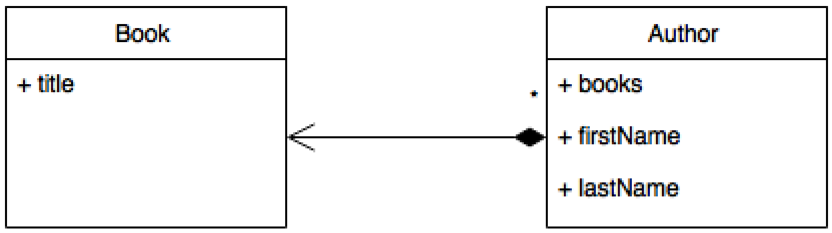
Object-relational mapping

Basic concept

Approach

- Basic idea
  - Class → Table
  - Field → Column
  - Relationships → FK

Object-oriented



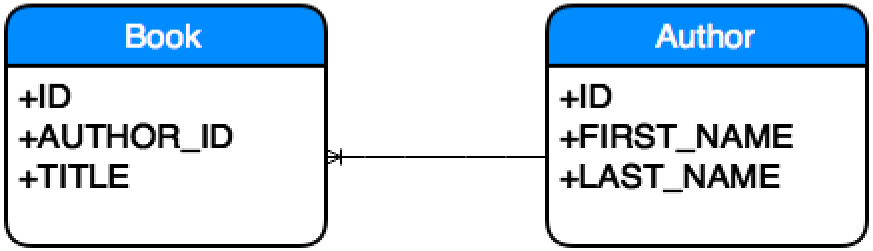
Book

+ title

Author

+ books
+ firstName
+ lastName

Relational schema



Book

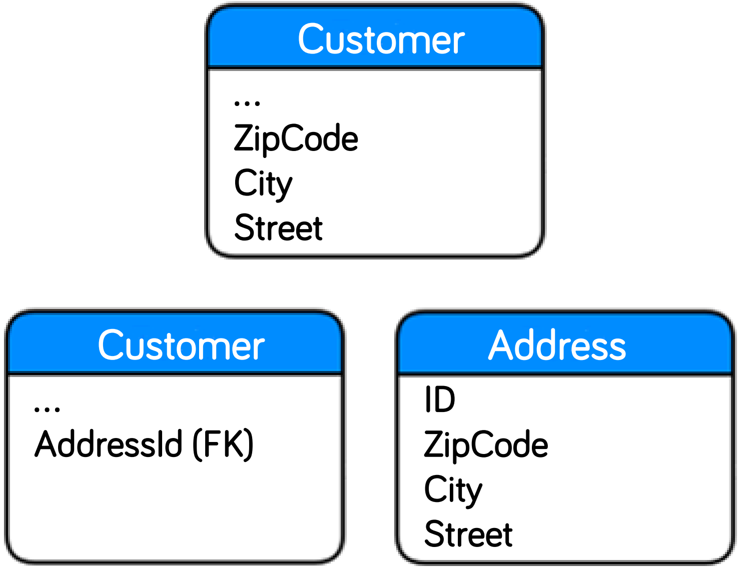
+ID
+AUTHOR_ID
+TITLE

Author

+ID
+FIRST_NAME
+LAST_NAME

Problems

- Compound fields
  - Customer
    - Address (Zip, City, Street)



Customer

...
ZipCode
City
Street

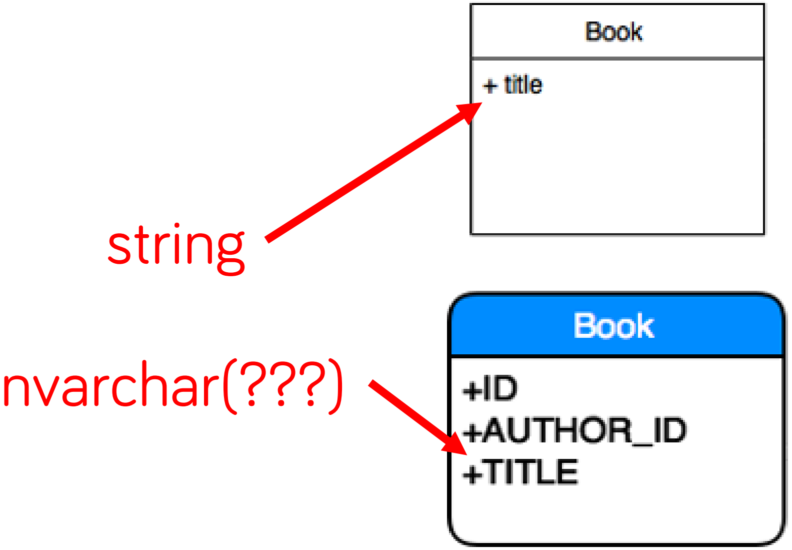
Customer

...
AddressId (FK)

Address

ID
ZipCode
City
Street

- Different data types
  - Conversion!



Book

+ title

string

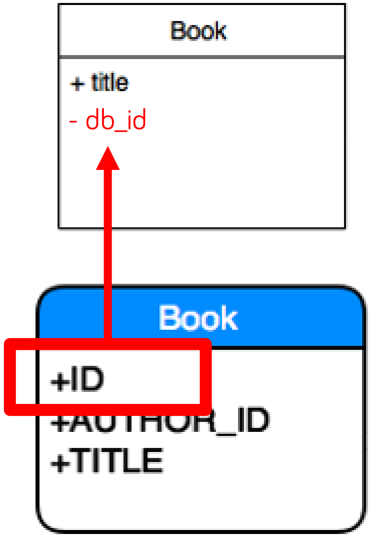
Book

+ID
+AUTHOR_ID
+TITLE

nvarchar(???)

Shadow information

- Required for persistency
  - Keys (primary keys)
  - Timestamps (for optimistic concurrency handling)
- Has no place in the business entity, but need to put it somewhere

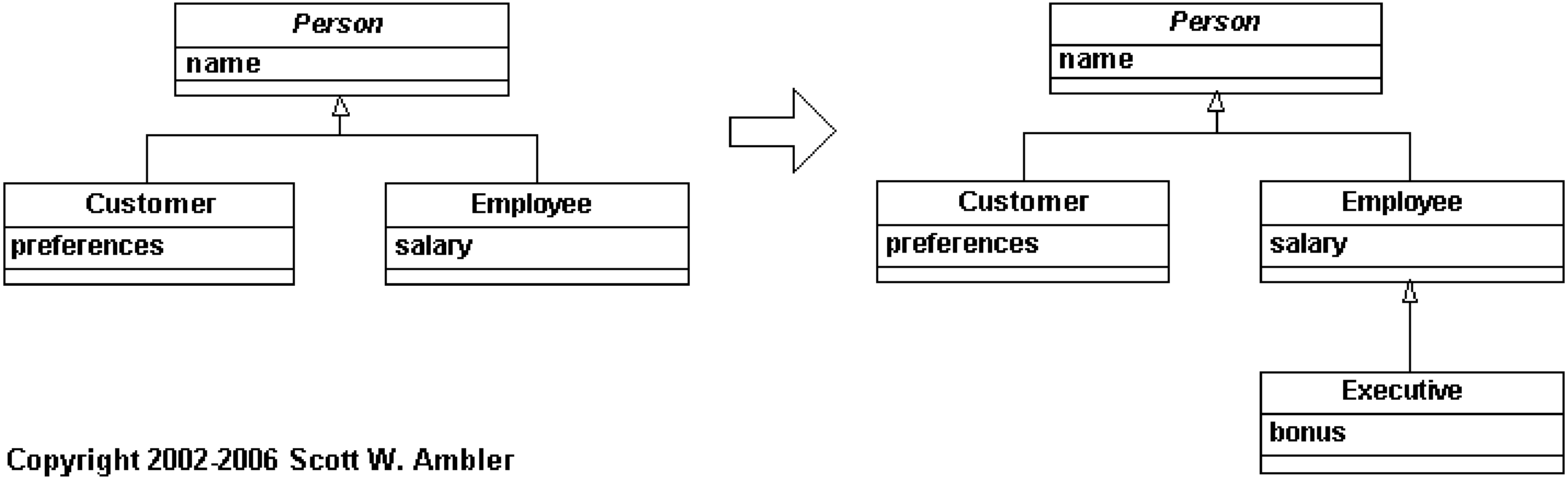


Object-relational mapping

Inheritance

Inheritance – Example

- Abstract class *Person*
- Multiple implementation
- New function built into the application -> new inheritance: *Executive*



Copyright 2002-2006 Scott W. Ambler

Inheritance

- Map the entire hierarchy into one table
- Map non-abstract types into their own table
  - From which objects can be instantiated
- Map all types into their own table
  - Abstract classes too
- General mapping independent of actual model

Mapping into a single table - 1

- List all attributes in the whole hierarchy
- Identify the type of the instance
  - Encoded in a new column
  - IsCustomer, IsEmployee,... columns
- Handling modification
  - Add new columns

Mapping into a single table - 2

Person table

ID	Name	Prefer.	Salary	IsCustomer	IsEmployee
1	X Y	NULL	1234	0	1
2	W Z	xxxxx	NULL	1	0

⇩ Introduce Executive

ID	Name	Prefer.	Salary	IsCustomer	IsEmployee	Bonus	IsExecutive
1	X Y	NULL	1234	0	1	NULL	0
2	W Z	xxxxx	NULL	1	0	NULL	0
3	Q Q	NULL	456	0	1	999	1

Mapping into a single table - 3

- Advantages
  - Simple
  - Easy to add a new inheritor
  - Easy to change the role of an instance
    - Employee→Executive
    - Employee and Customer too
- Drawbacks
  - Wasting storage space
  - Change in a single class changes storage model of all
  - Hard to overview in case of a complex structure
- Use when … - *design time* dimension
  - Simple hierarchy

Mapping real classes - 1

- Unique table per class
- All attributes of the class
- Instance identifier
- Handling change
  - New class → new table
  - Change in an attribute → Must be propagated in the hierarchy

Mapping real classes - 2

Customer table

ID	Name	Prefer.
2	W Z	xxxxx

Employee table

ID	Name	Salary
1	X Y	1234

⇩ Introduce Executive

Executive table

ID	Name	Salary	Bonus
4	QQ	456	999

Mapping real classes - 3

- Advantages
  - Intuitive
  - Better fit to the OO paradigm
  - Fast data access
- Drawbacks
  - In case a class is modified → Might affect multiple tables
  - Instances that have multiple roles
    - Employee →Executive
    - Employee and Customer at the same time
- Use when … - *design time* dimension
  - Structure that changes infrequently

Mapping all classes - 1

- Tables match the inheritance hierarchy
- Map parent-child relationship with foreign keys
- Instance idendifier

Mapping all classes - 2

Person table

ID	Name
1	X Y
2	W Z

Customer table

CID	PID (FK)	Prefer.
11	2	xxxxx

Employee table

EID	PID (FK)	Salary
21	1	1234

⇩ Introduce Executive

Executive table

XID	EID (FK)	Bonus
41	22	999

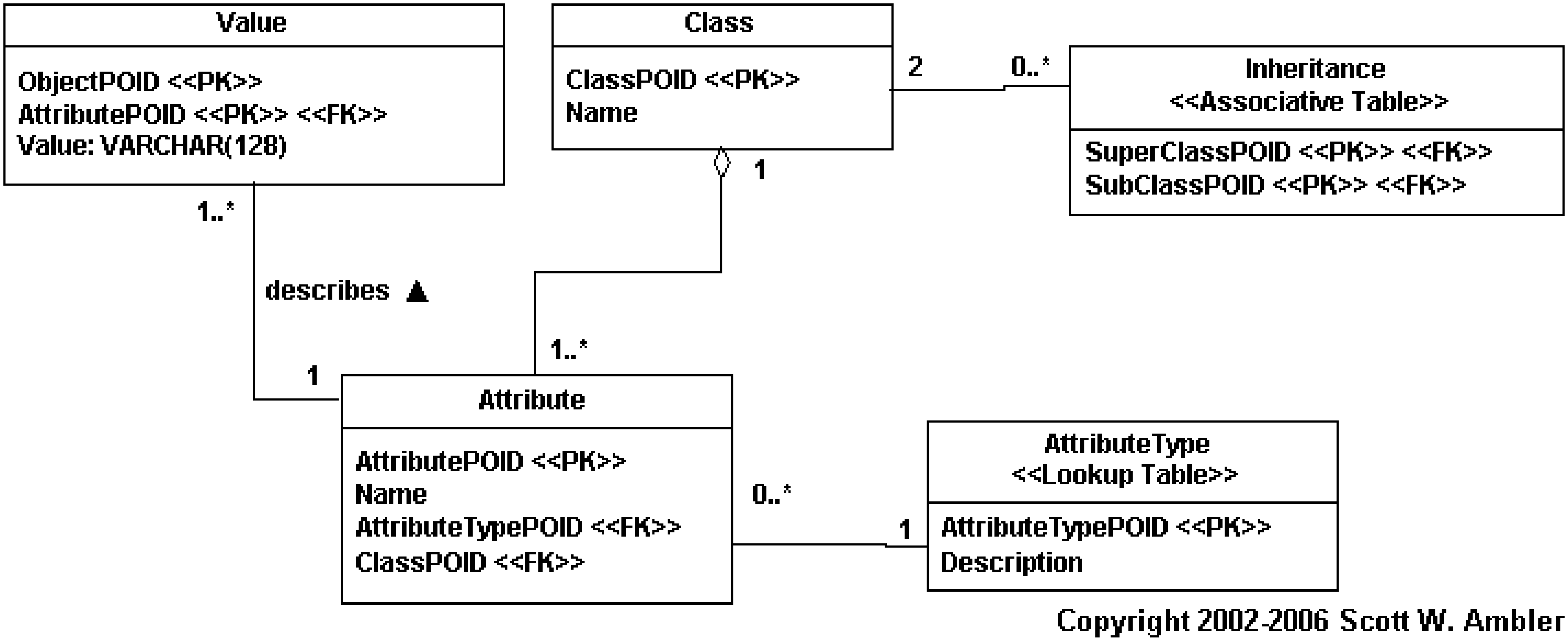
Mapping all classes - 3

- Advantages
  - Intuitive
  - Easy to modify parent class structure
- Drawbacks
  - Complex database scheme
  - Data of a single instance is scattered in multiple tables
    - Requires more complex queries
    - Join is needed → slower
- Use when … - *design time* dimension
  - Complex hierarchies
  - Structure that changes frequently

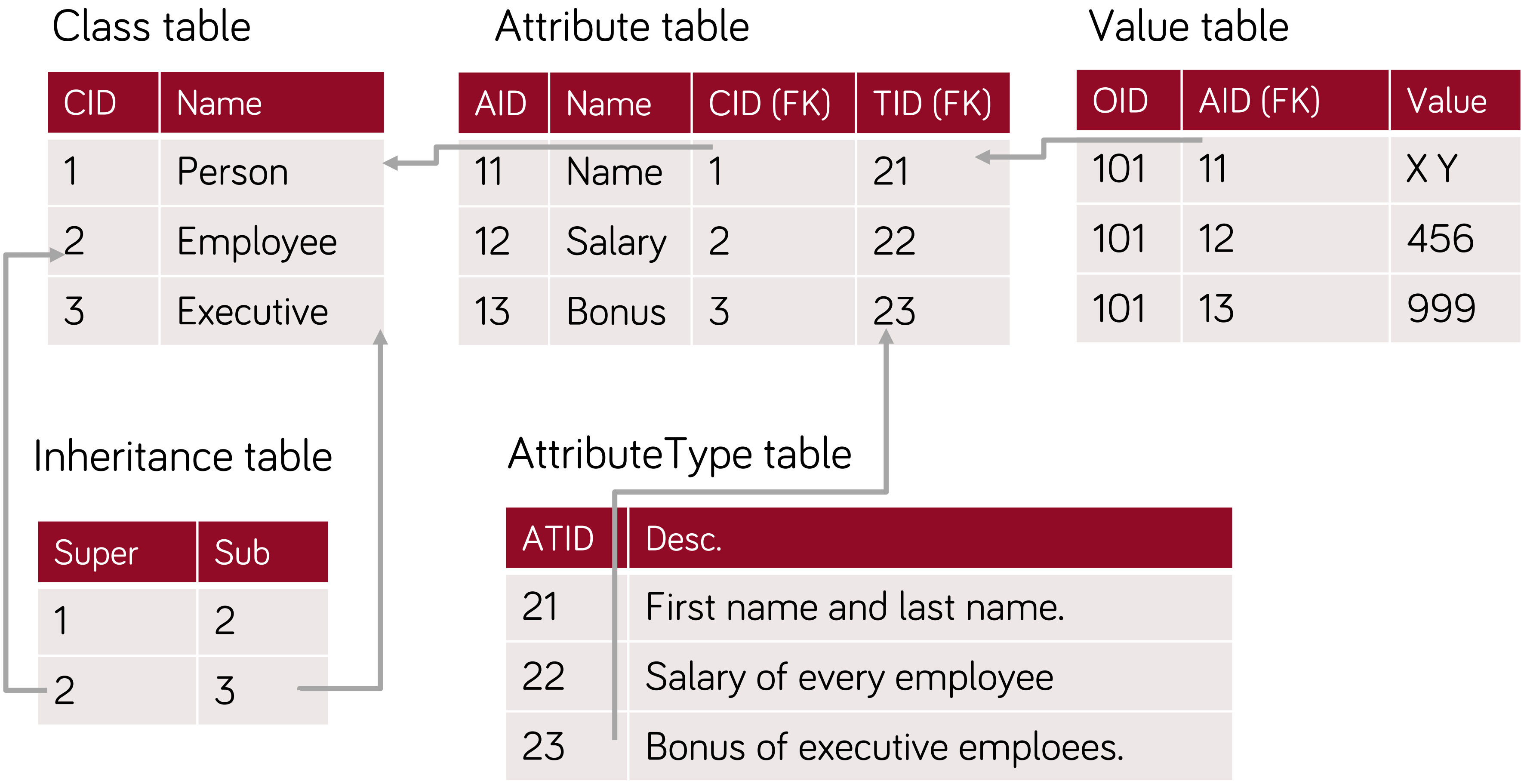
Mapping into a general scheme - 1

- Meta data driven solution
- General scheme
  - Can represent any hierarchy
  - Independent of the specific classes
    - Class hierarchy →Metadata
    - Class instances → Attributes

Mapping into a general scheme - 2



Mapping into a general scheme - 3



Class table

CID	Name
1	Person
2	Employee
3	Executive

Attribute table

AID	Name	CID (FK)	TID (FK)
11	Name	1	21
12	Salary	2	22
13	Bonus	3	23

Value table

OID	AID (FK)	Value
101	11	X Y
101	12	456
101	13	999

Inheritance table

Super	Sub
1	2
2	3

AttributeType table

ATID	Desc.
21	First name and last name.
22	Salary of every employee
23	Bonus of executive emploees.

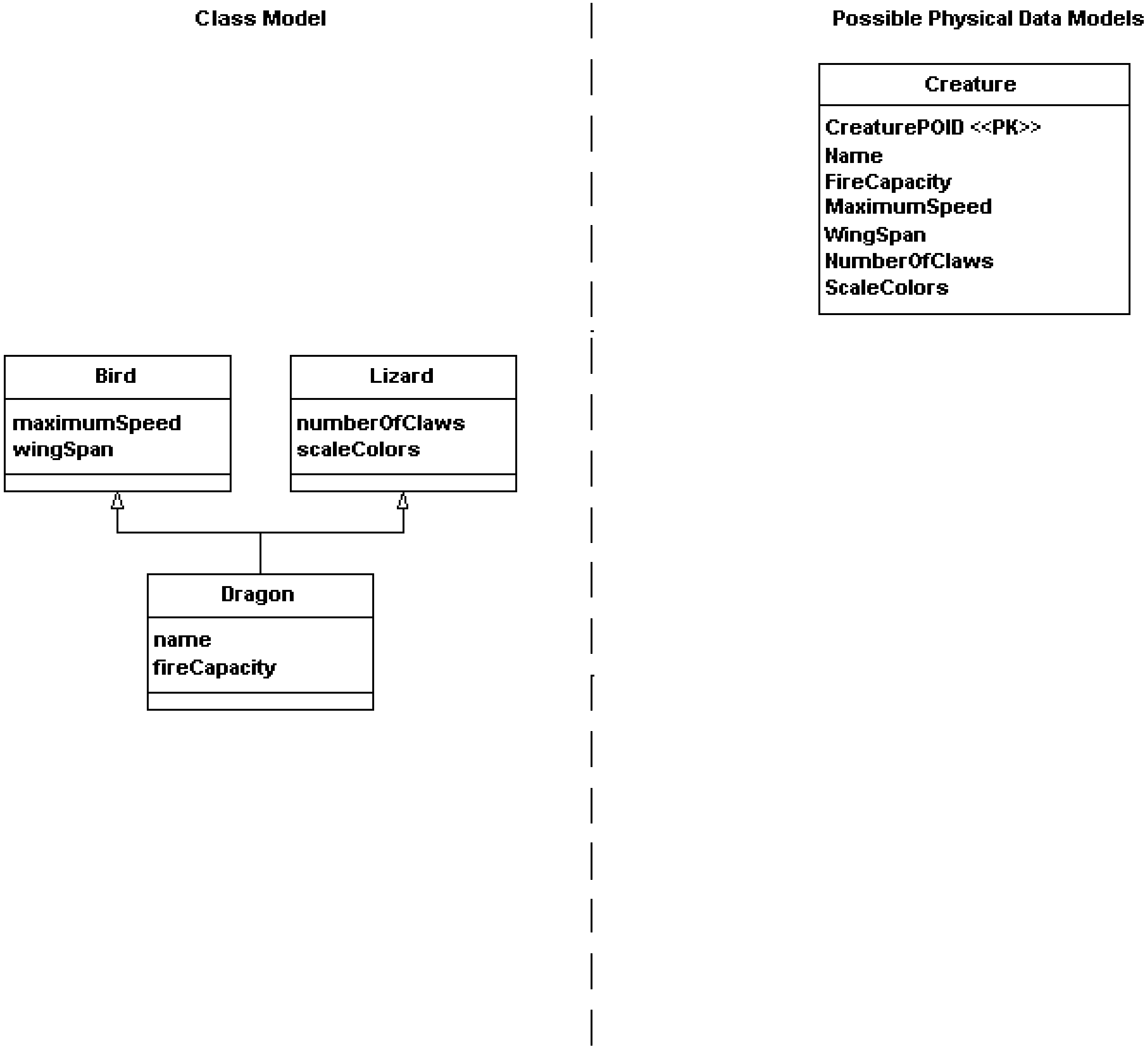
Mapping into a general scheme - 4

- Advantages
  - Flexible
  - Can describe anything
- Drawbacks
  - Hard to understand at first
  - Hard to gather all data of an instance
  - Not efficient for large data sets
- Use when … - *design time* dimension
  - Complex applications
  - Small amount of data
  - Can change in runtime

Multiple inheritance

- Modern languages do not support
  - But C++ → Still supported
- Same solutions are applicable
  - The general solution includes this case

Multiple inheritance



Class Model

Bird
maximumSpeed
wingSpan

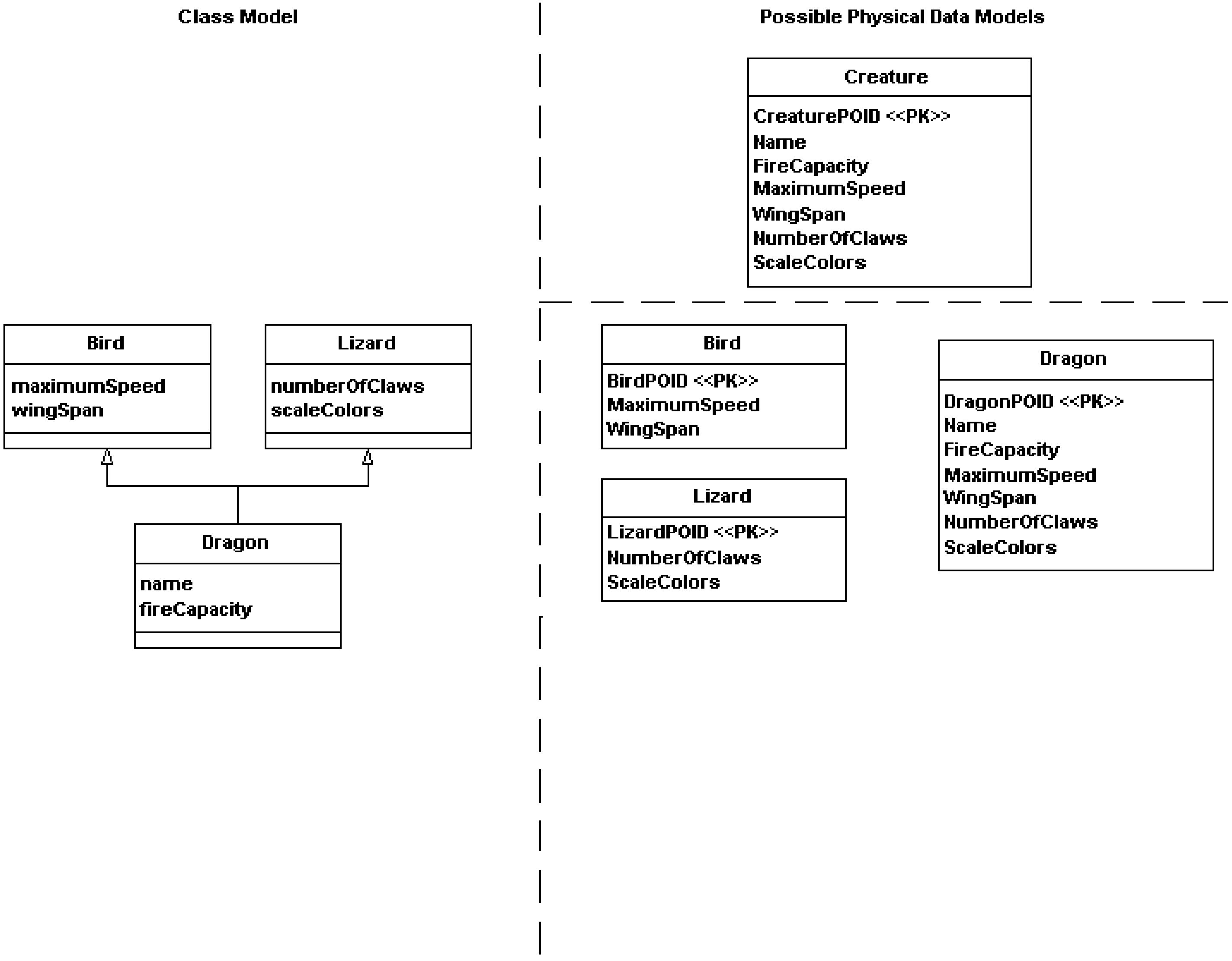
Lizard
numberOfClaws
scaleColors

Dragon
name
fireCapacity

Possible Physical Data Models

Creature
CreaturePOID <<PK>>
Name
FireCapacity
MaximumSpeed
WingSpan
NumberOfClaws
ScaleColors

Multiple inheritance



Class Model

Bird
maximumSpeed
wingSpan

Lizard
numberOfClaws
scaleColors

Dragon
name
fireCapacity

Possible Physical Data Models

Creature
CreaturePOID <<PK>>
Name
FireCapacity
MaximumSpeed
WingSpan
NumberOfClaws
ScaleColors

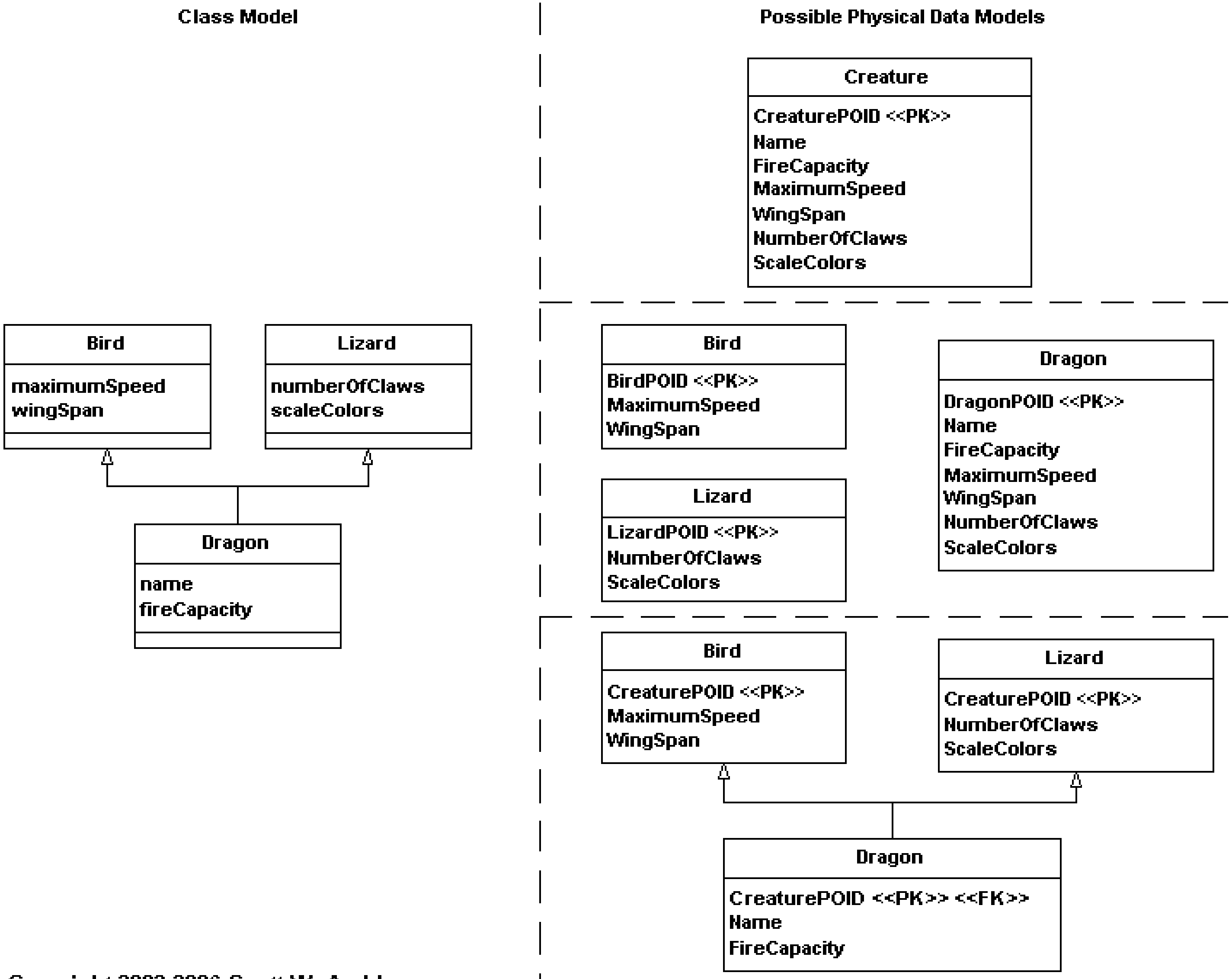
Bird
BirdPOID <<PK>>
MaximumSpeed
WingSpan

Lizard
LizardPOID <<PK>>
NumberOfClaws
ScaleColors

Dragon
DragonPOID <<PK>>
Name
FireCapacity
MaximumSpeed
WingSpan
NumberOfClaws
ScaleColors

Copyright 2002-2006 Scott W. Ambler

Multiple inheritance



Class Model

Bird

maximumSpeed
wingSpan

Lizard

numberOfClaws
scaleColors

Dragon

name
fireCapacity

Possible Physical Data Models

Creature

CreaturePOID <<PK>>
Name
FireCapacity
MaximumSpeed
WingSpan
NumberOfClaws
ScaleColors

Bird

BirdPOID <<PK>>
MaximumSpeed
WingSpan

Lizard

LizardPOID <<PK>>
NumberOfClaws
ScaleColors

Dragon

DragonPOID <<PK>>
Name
FireCapacity
MaximumSpeed
WingSpan
NumberOfClaws
ScaleColors

Bird

CreaturePOID <<PK>>
MaximumSpeed
WingSpan

Lizard

CreaturePOID <<PK>>
NumberOfClaws
ScaleColors

Dragon

CreaturePOID <<PK>> <<FK>>
Name
FireCapacity

Which one to use?

- **The above mentions design-time aspects are often secondary!**
- What queries and actions do we perform and how often?
  - Filters, orders etc.
- What amount of data do we have?
- What is the distribution of the data?
- You even have to test and measure the alternatives!

Object-relational mapping

Types of relationships

Object Relationships – 1

- Types of relationships
  - Association
  - Aggregation
  - Composition
- Types
  - One to one
  - One to many
  - Many to many

→ Referential integrity

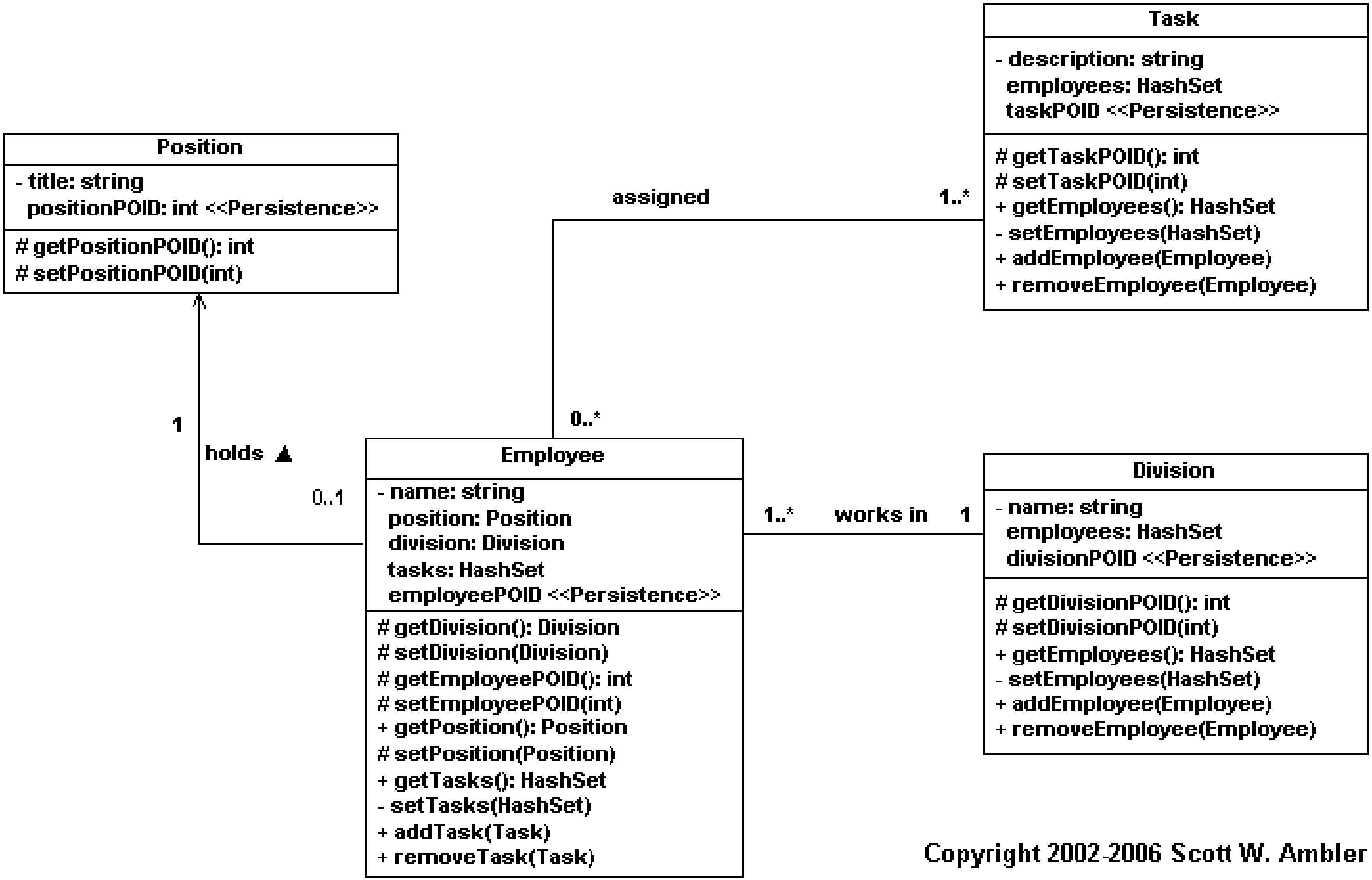
- Directed
  - Oneway
  - Navigable both ways

→ Cannot be mapped

Object Relationships – 2

- One to one
  - Foreign key in one of the tables
    - (May imply one to many connection)
- One to many
  - Foreign key to the “one”
- Many to many
  - Cannot be mapped directly
  - Requires a junction table
- Cardinality
  - Both ends required - do not map
  - Hard to start populating the dataset
  - Cardinalities 0, 1, many

Object Relationships – 3



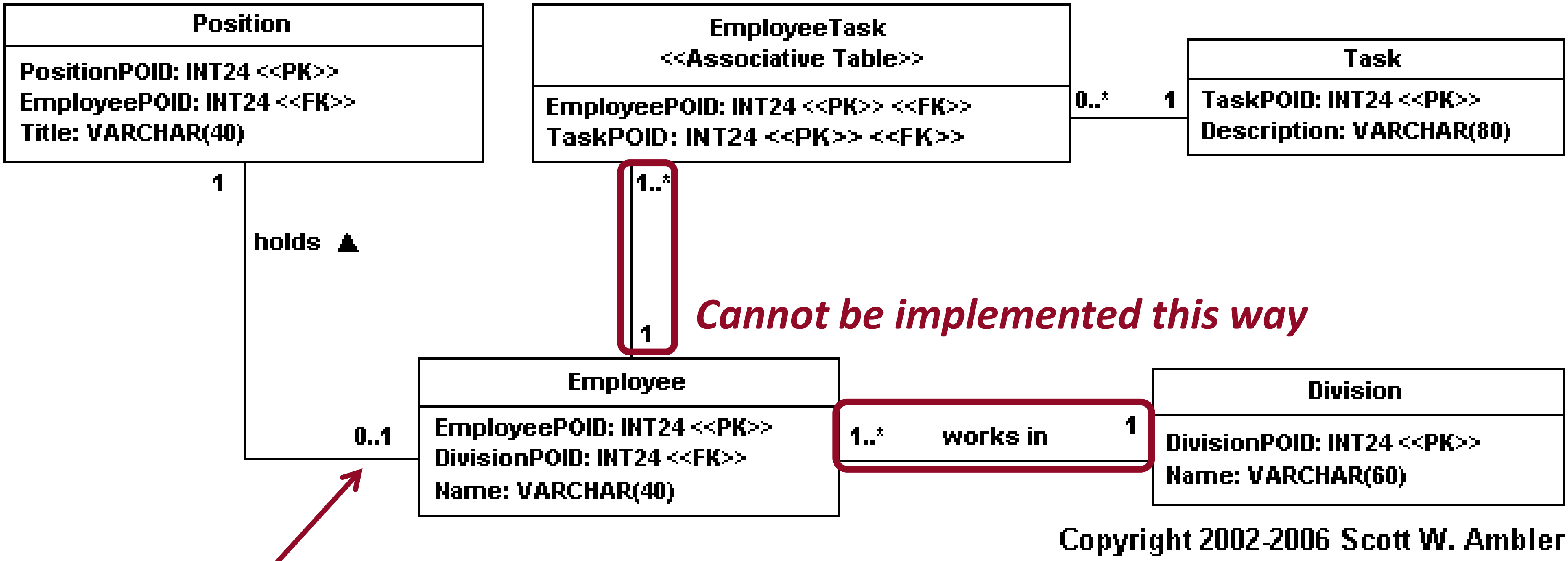
Task
- description: string
employees: HashSet
taskPOID <<Persistence>>
# getTaskPOID(): int
# setTaskPOID(int)
+ getEmployees(): HashSet
- setEmployees(HashSet)
+ addEmployee(Employee)
+ removeEmployee(Employee)

Position
- title: string
positionPOID: int <<Persistence>>
# getPositionPOID(): int
# setPositionPOID(int)

Employee
- name: string
position: Position
division: Division
tasks: HashSet
employeePOID <<Persistence>>
# getDivision(): Division
# setDivision(Division)
# getEmployeePOID(): int
# setEmployeePOID(int)
+ getPosition(): Position
# setPosition(Position)
+ getTasks(): HashSet
- setTasks(HashSet)
+ addTask(Task)
+ removeTask(Task)

Division
- name: string
employees: HashSet
divisionPOID <<Persistence>>
# getDivisionPOID(): int
# setDivisionPOID(int)
+ getEmployees(): HashSet
- setEmployees(HashSet)
+ addEmployee(Employee)
+ removeEmployee(Employee)

Object Relationships – 4



Cannot be implemented this way

Copyright 2002-2006 Scott W. Ambler

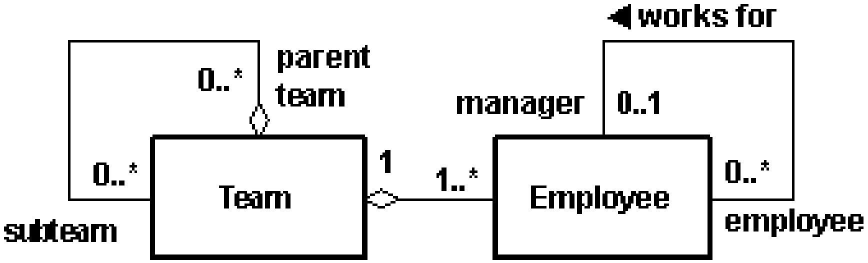
Could be one to many based on the structure

Recursion – 1

- Also known as reflection
- Both ends of the relationship terminate at the same class
- Like all the other relationships
  - Little difference in many to many

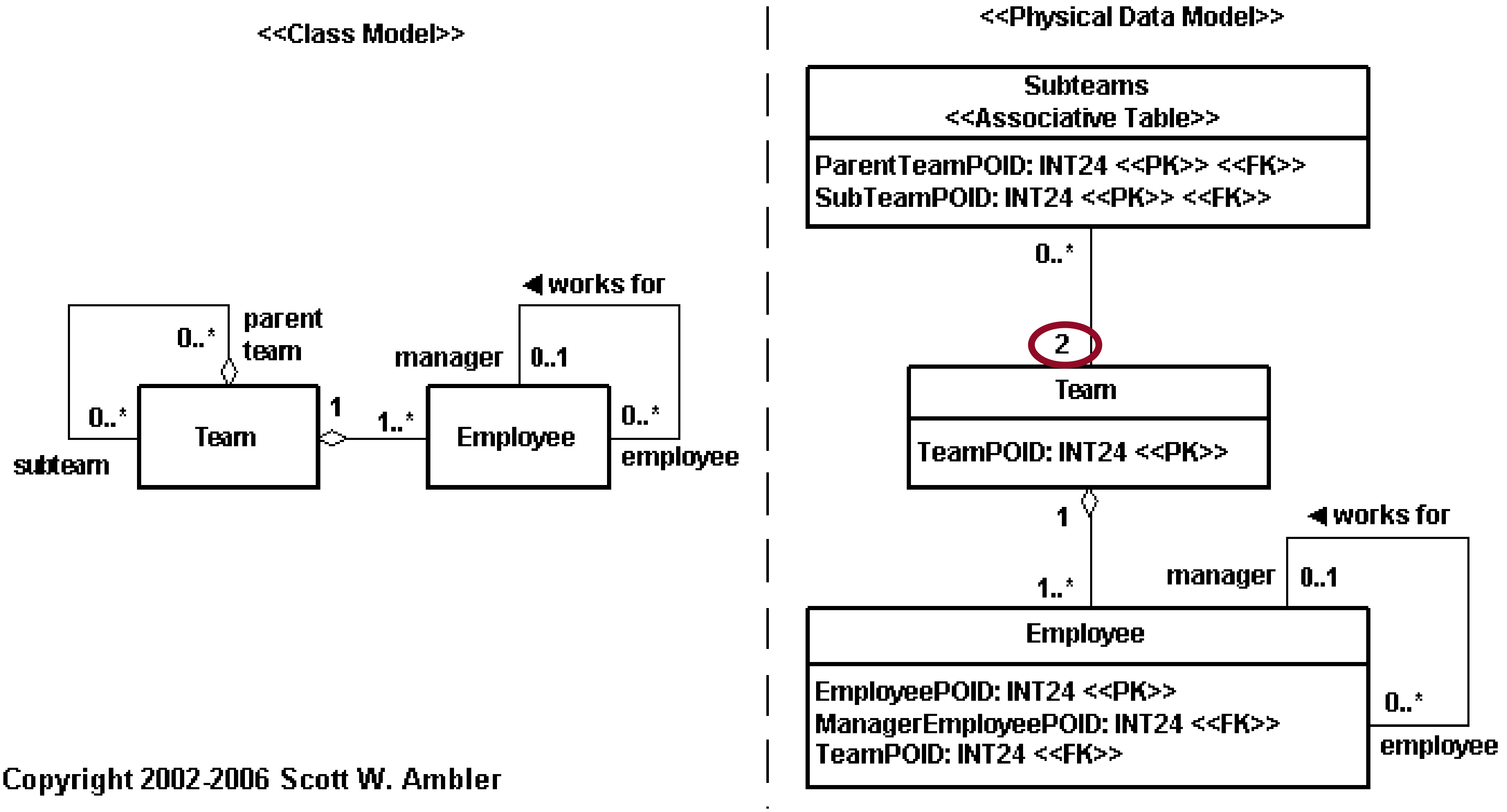
Recursion – 2

<<Class Model>>



Copyright 2002-2006 Scott W. Ambler

Recursion – 2

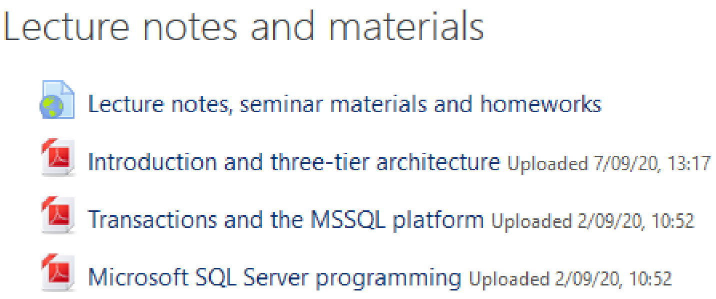


<<Class Model>>
0..*
parent
team
0..*
subteam
Team
1
1..*
◀ works for
manager
0..1
Employee
0..*
employee
<<Physical Data Model>>
Subteams
<<Associative Table>>
ParentTeamPOID: INT24 <<PK>> <<FK>>
SubTeamPOID: INT24 <<PK>> <<FK>>
0..*
2
Team
TeamPOID: INT24 <<PK>>
1
1..*
◀ works for
manager
0..1
Employee
EmployeePOID: INT24 <<PK>>
ManagerEmployeePOID: INT24 <<FK>>
TeamPOID: INT24 <<FK>>
0..*
employee

Object-relational mapping

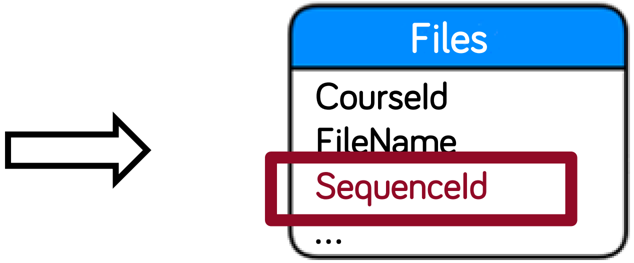
Further problems

Ordered collection



Lecture notes and materials

- Lecture notes, seminar materials and homeworks
- Introduction and three-tier architecture Uploaded 7/09/20, 13:17
- Transactions and the MSSQL platform Uploaded 2/09/20, 10:52
- Microsoft SQL Server programming Uploaded 2/09/20, 10:52



- Attribute the order is based on + read from the DB according to this
- Change in the order may effect
  - Change in multiple records
  - Alternative: empty intervals in-between
- Deletion
  - Not always required to re-index
    - Meaning is sort ordering, not the actual position

Class-level properties - 1

- Specific to the class
- Does not relate to any instance of the class
- Example
  - Identifier of the next invoice
  - Fixed percentage of sale today
- Class-level constants

Class-level properties - 2

- Each property into a new table
  - Fast
  - Lot of small tables → chaotic data model

InvoiceGlobals table

NextId
123

Sale table

Sale
2%

Class-level properties - 3

- Single table, all properties into specific columns
  - Fast
  - Simple (only a single table)
  - Problematic concurrency (since this is column and not row-based)

Globals table

NextInvoiceId	Sale
123	2%

Class-level properties - 4

- Single table per class
  - Fast
  - Lot of small tables → chaotic data model

InvoiceGlobals table

NextId	PaymentDue
123	30

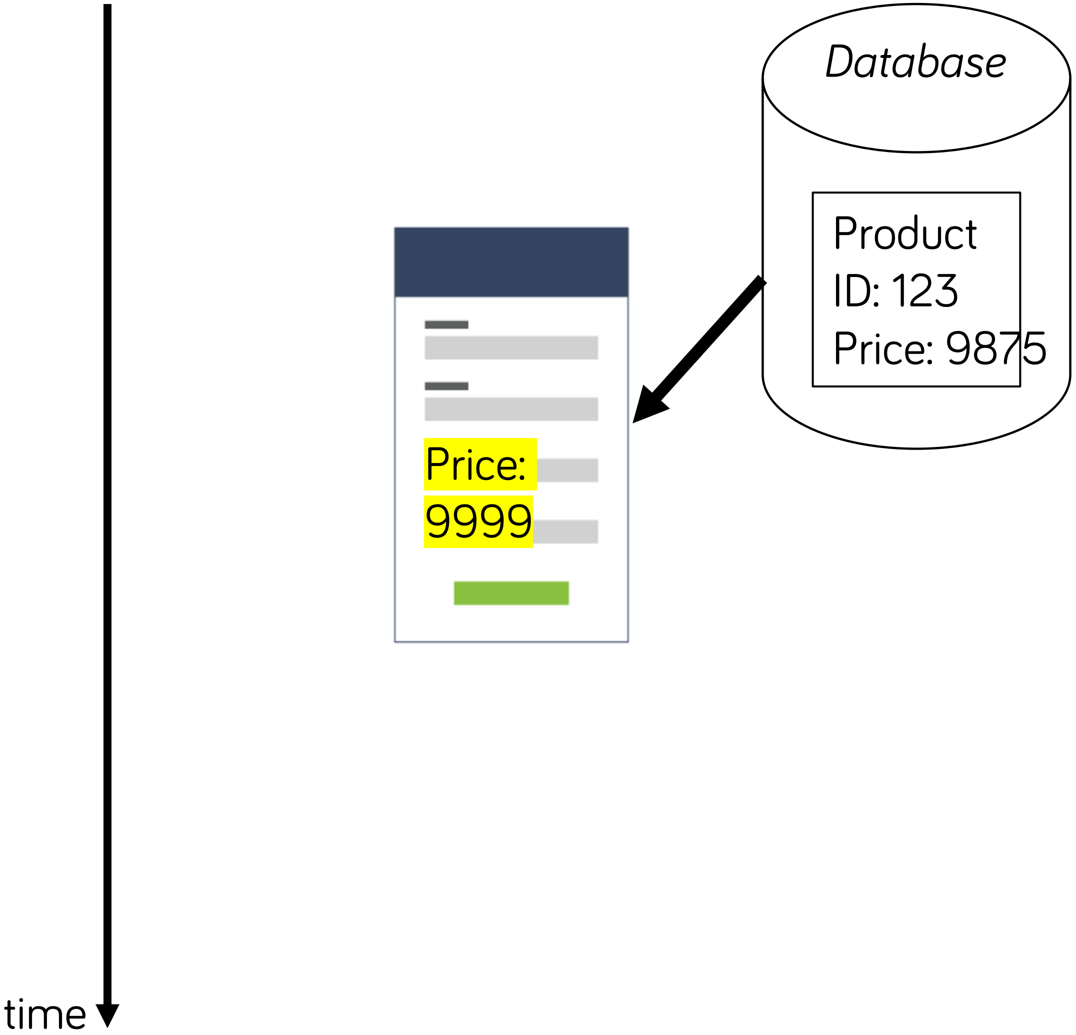
Class-level properties - 5

- General solution
  - Each property a new row
    - Class
    - Name of the property
    - Value
  - Have to convert the data
  - Easy to extend
    - New property → New record

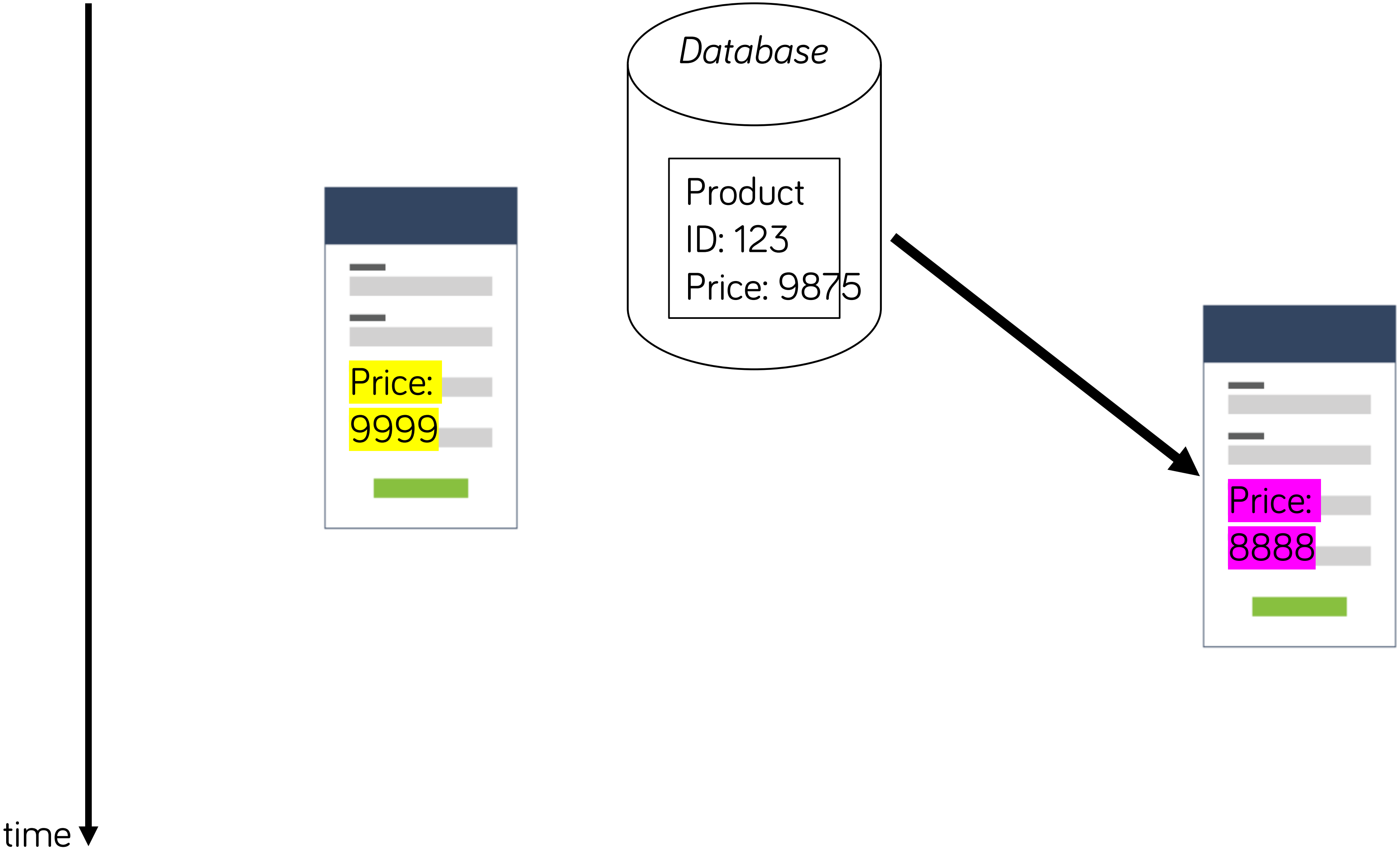
Class	Key	Value
Invoice	NextId	123
Invoice	Sale	2

Concurrency handling

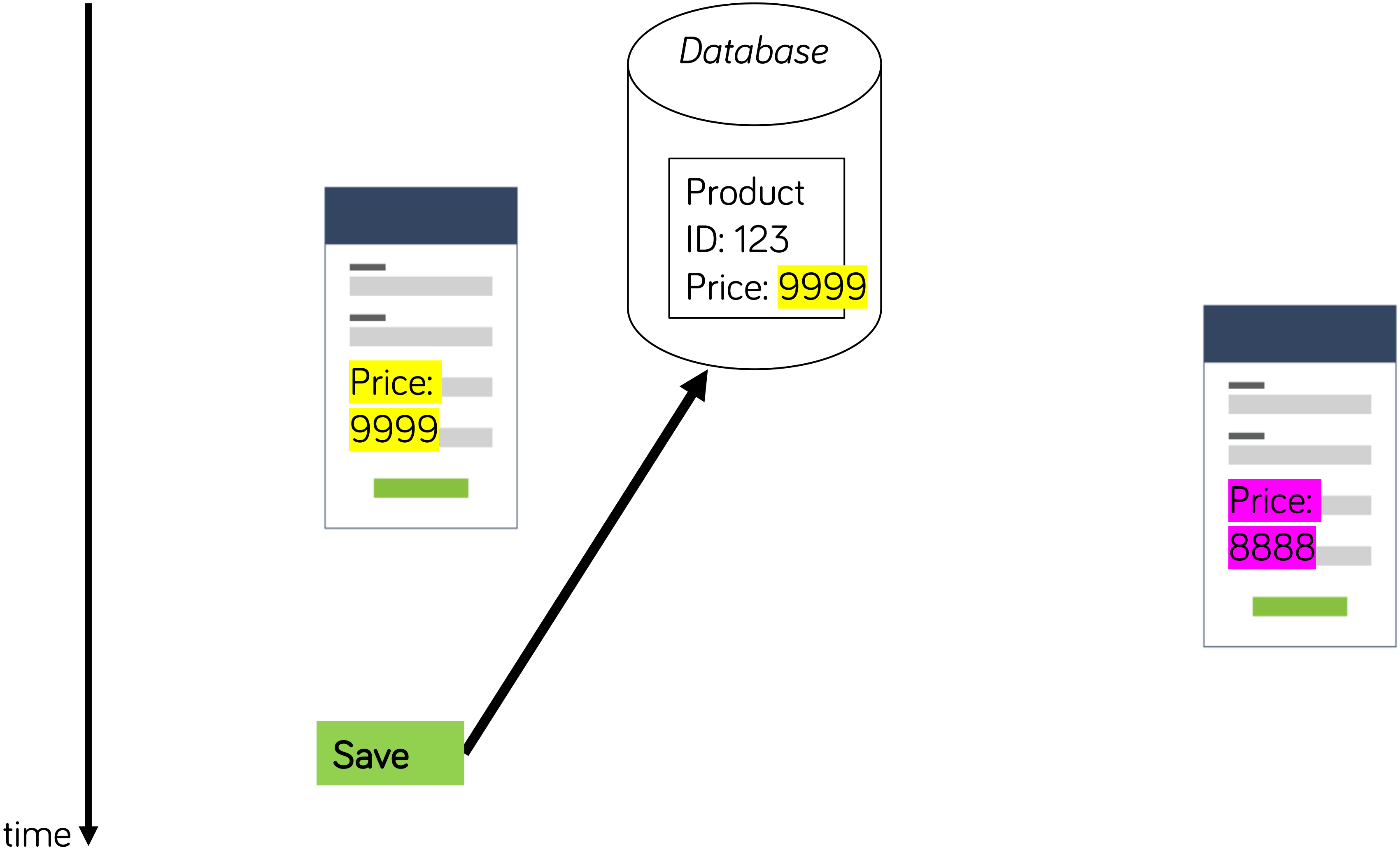
Concurrency handling: problem



Concurrency handling: problem



Concurrency handling: problem



Database

Product
ID: 123
Price: 9999

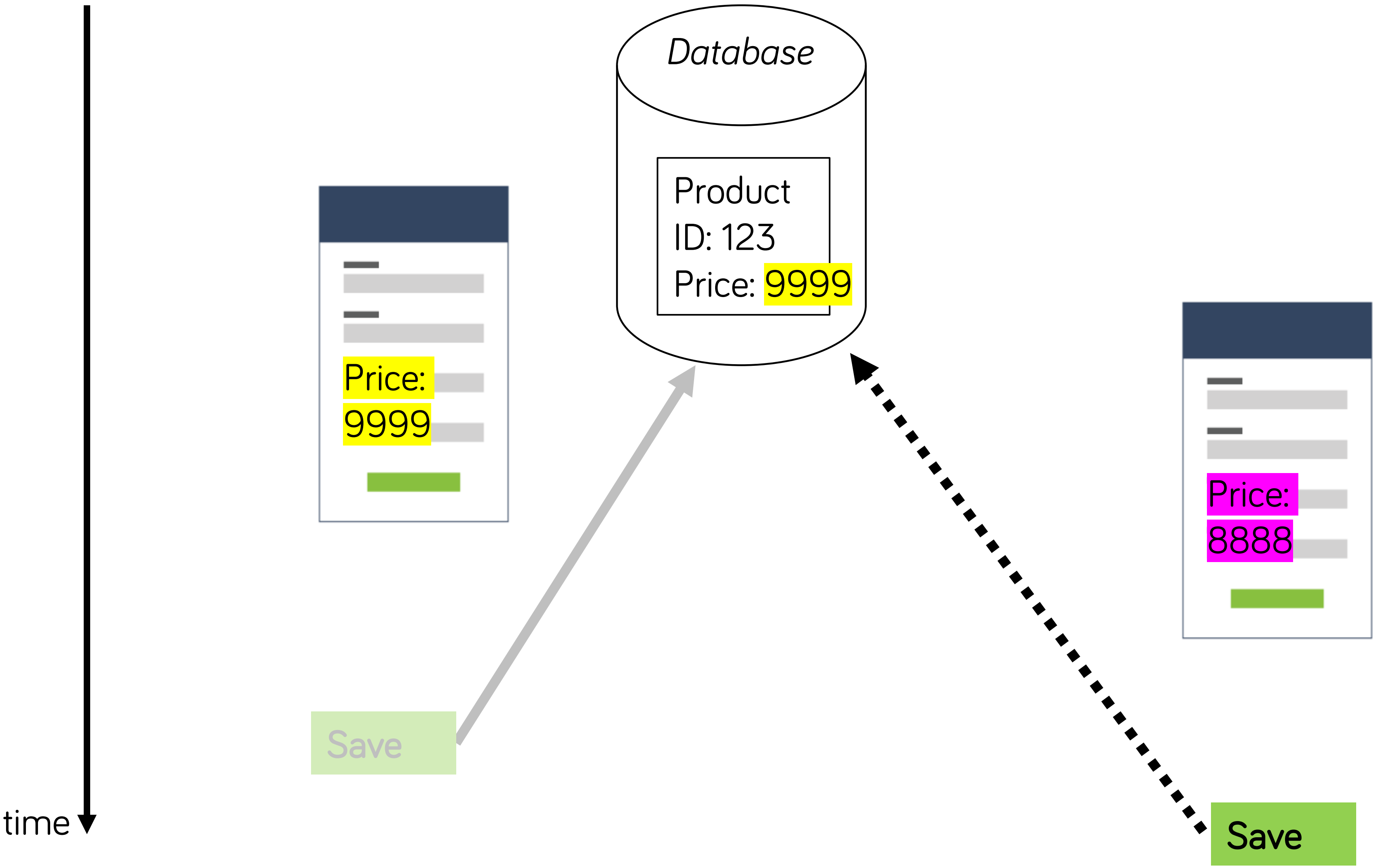
Price:
9999

Price:
8888

Save

time

Concurrency handling: problem



Concurrency handling

- None – seldom a good solution ☹
- Pessimistic concurrency handling
  - Someone else is going to modify the same data
  - Get exclusive access to the data
    - Locking + Transactions
- Optimistic
  - No one will modify
  - But we must detect clash

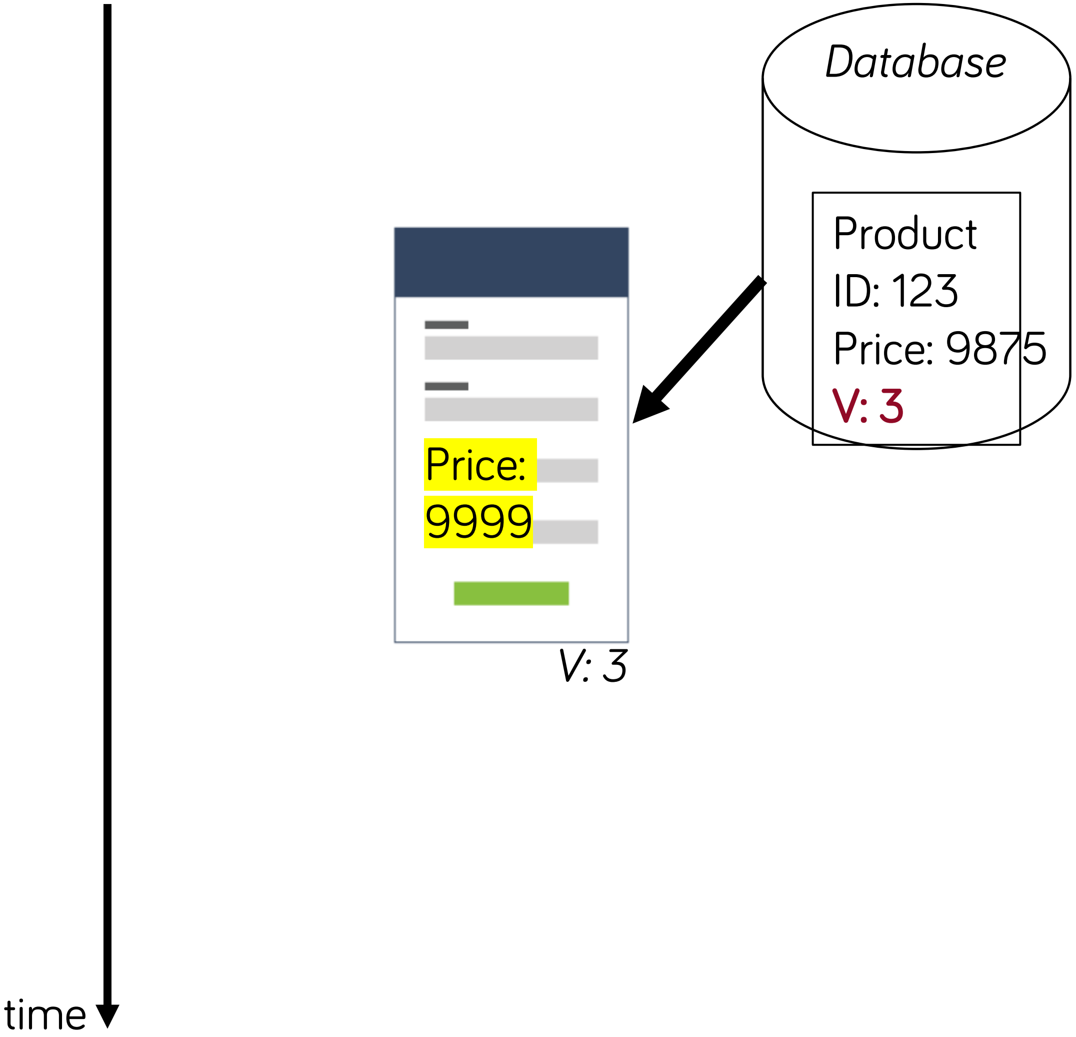
Pessimistic concurrency handling

- When a connection with the database is maintained
  - Locking individual records
  - Database transactions do this
- If the connection to the database has been closed already (UI)
  - Typically happens in the Business Logic Layer
  - Exclusion handled manually in the Business Logic Layer
  - Scalability + SPOF
  - Handling closed session

Optimistic concurrency handling

- Must check changes before committing them
- Compare record contents
  - Record versioning
    - Logical counter
    - Requires database scheme change
  - Compare the contents
    - Preserve the state before the changes
    - Additional information within the business entities
- The updating SQL command must be written in a specific way
  - Data access layer provides this

Optimistic concurrency handling



Database

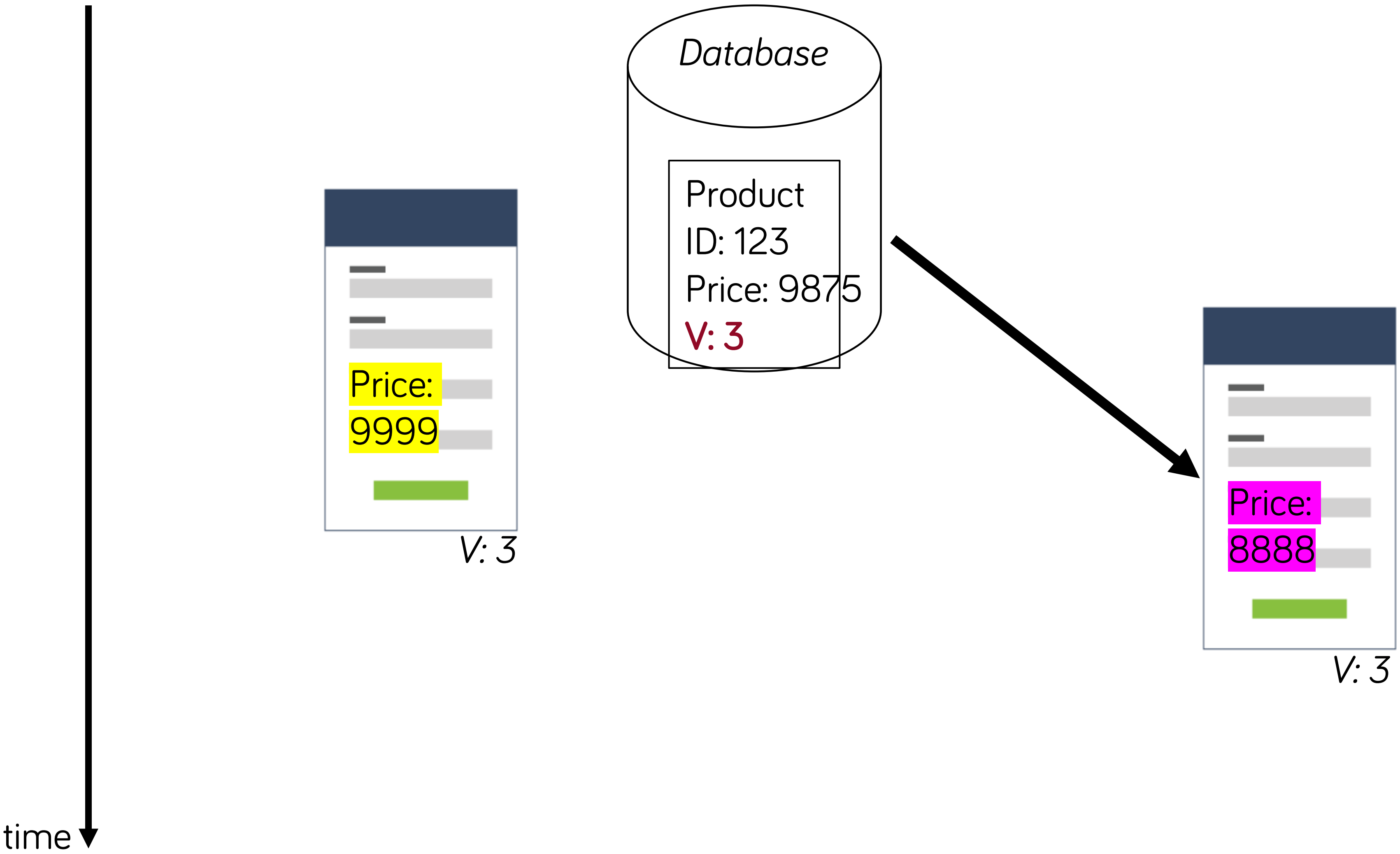
Product
ID: 123
Price: 9875
V: 3

Price:
9999

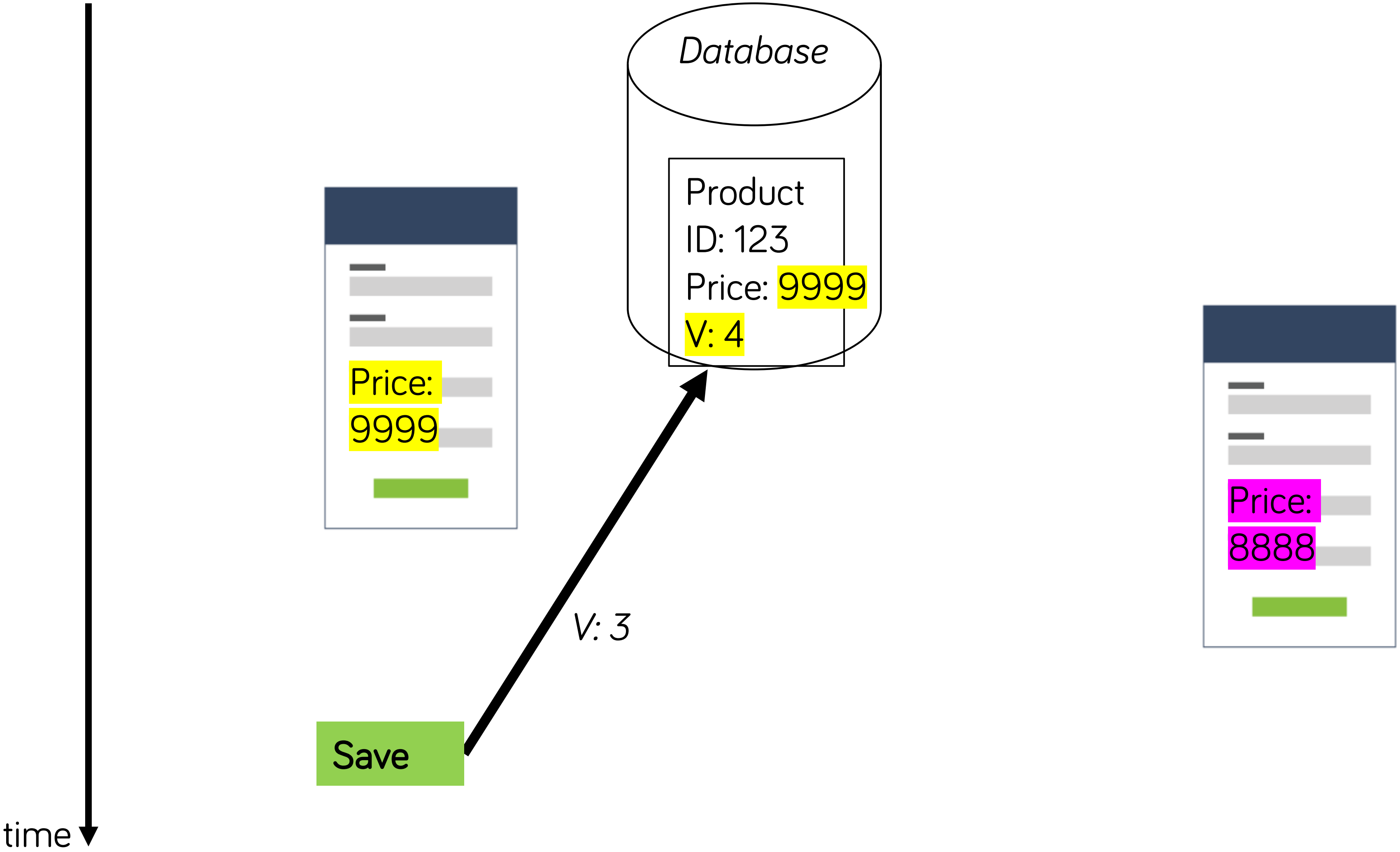
V: 3

time

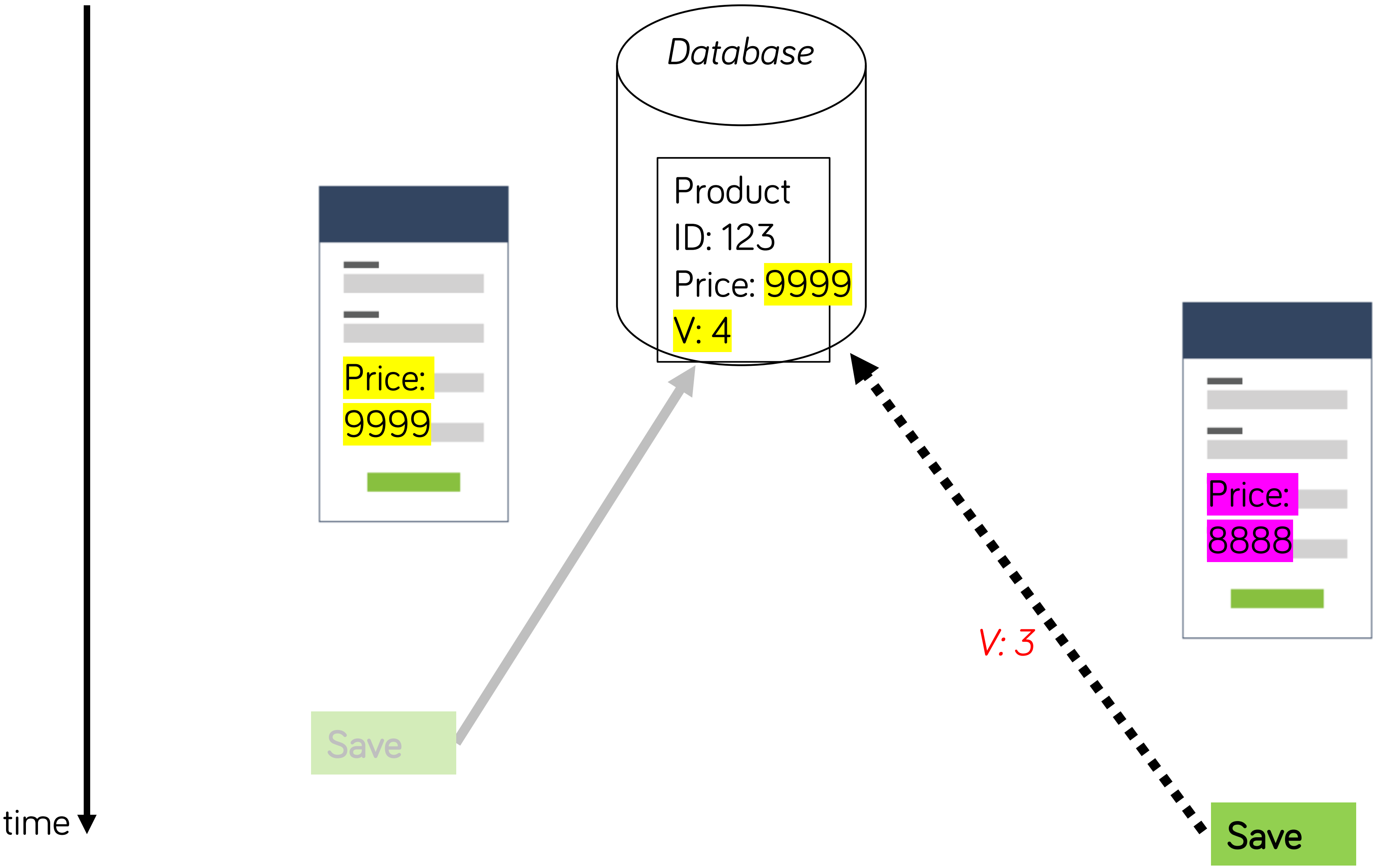
Optimistic concurrency handling



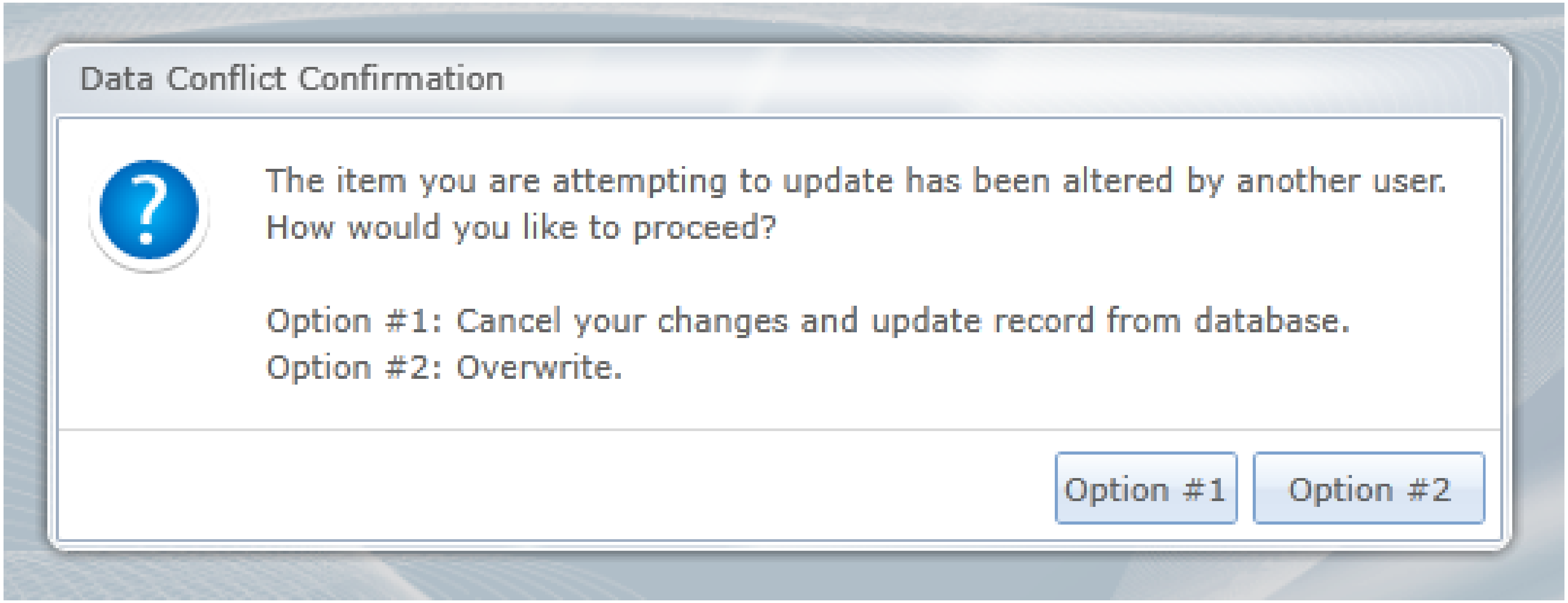
Optimistic concurrency handling



Optimistic concurrency handling



Optimistic concurrency handling: resolution



Data Conflict Confirmation

The item you are attempting to update has been altered by another user. How would you like to proceed?

Option #1: Cancel your changes and update record from database.
Option #2: Overwrite.

Image source: http://reptonprojects.com/index.php/application/concurrency-control

Detection strategies

1. **The first writer wins** 😃
2. Last writer wins 🙄
3. We leave it to the user 😱
4. We compare the set of modified fields 🧐
   - Consistency must be maintained!
5. In the vast majority of cases, the changes do not only affect a single table, but several records! 😳 😵

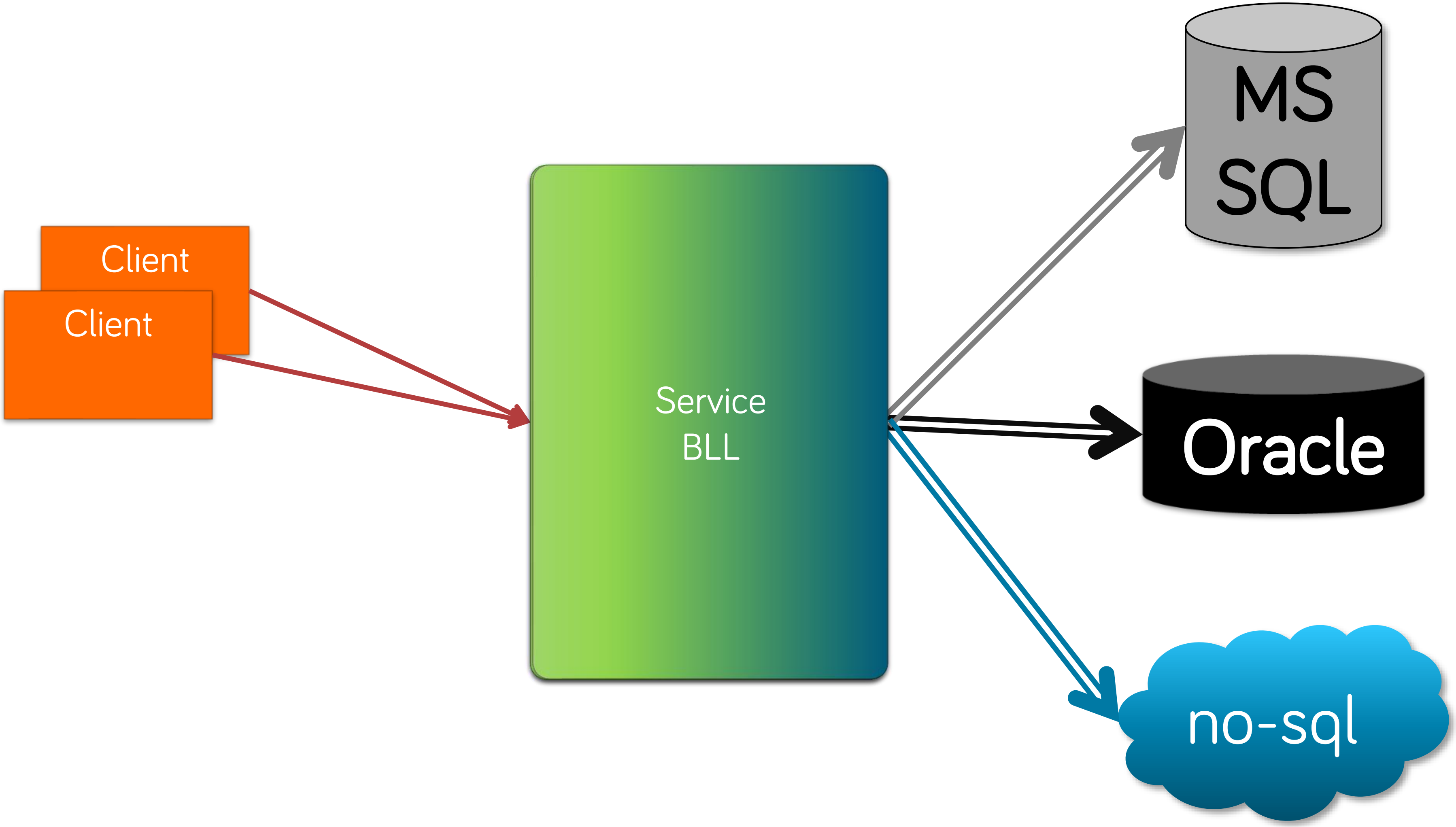
Detection -> API

- In the case of a multi-layered application, the data is transferred to the UI layer
- The database connection is closed
- In case of a modification request, it is necessary to know **from somewhere** what the original status / version of the records were
- The server is stateless -> the UI must store information about the original record
  - **-> the API changes**

Design patterns in the data-access layer

Repository

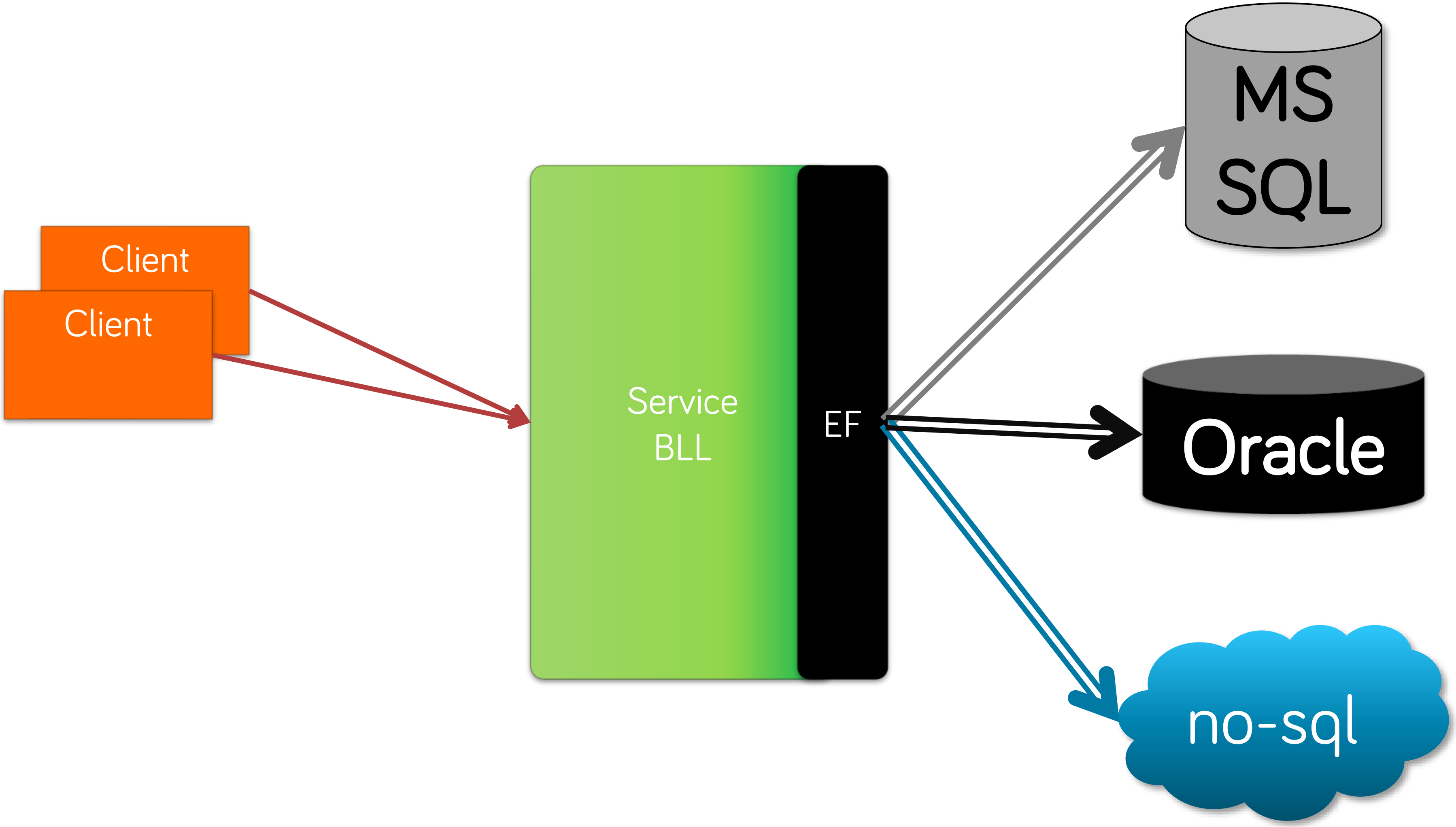
Multiple data storage solutions



Solution

- Place the parts to be replaced behind an interface!
- Loose coupling
- It is clearly visible what needs to be rewritten
- It can be tested
- Tasks can be better divided between teams/devs

Multiple data storage solutions



Advantages of data access layer separation

- Data access layer replacement
  - Linq to EF, nHibernate, …
- Because of unit test, for mocking
- Database refactoring
- You need to switch to another technology because it generates better SQL for the database engine
- It is necessary to switch to another database that is not or poorly supported by the current provider
- …

Is it necessary to replace it?

- YAGNI: You aren’t gonna need it
  - "do the simplest thing that could possibly work“
- KISS: Keep it simple and stupid

- **Agile:** don't plan and implement the system based on *possible future* requirements!
  - Customer is paying when he wants it...
- **Waterfall**: think hard about whether this kind of flexibility will be needed during the application's lifetime
  - If so, can the costs be passed on?

Alternative solutions

- What is the exact reason why this might be necessary?

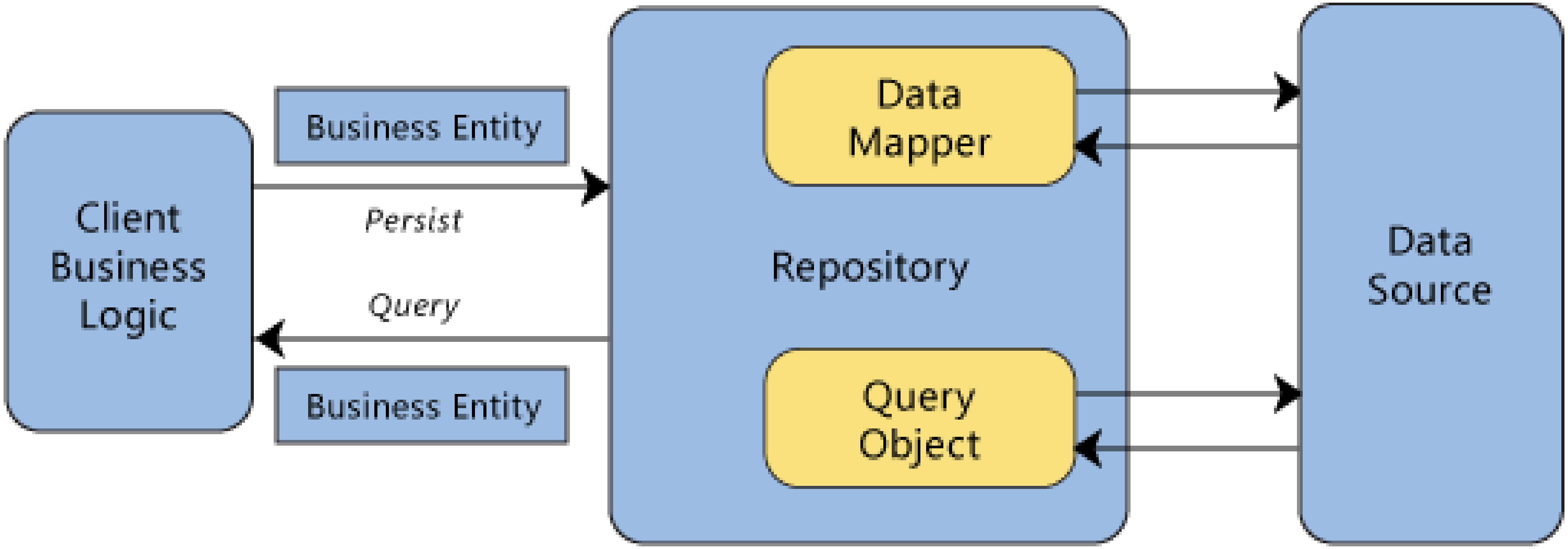
- Unit testing?
  - Search in-memory database, SQLite, …
- Replacing the database?
  - Look for paid components, see if there is a provider
- To what extent should the data layer be hidden from the business logic?

Separation of data (access) layer

- Separation: typical interface design
- The services and functions of the lower layer are explicitly exposed on the interface
- A widely used solution:

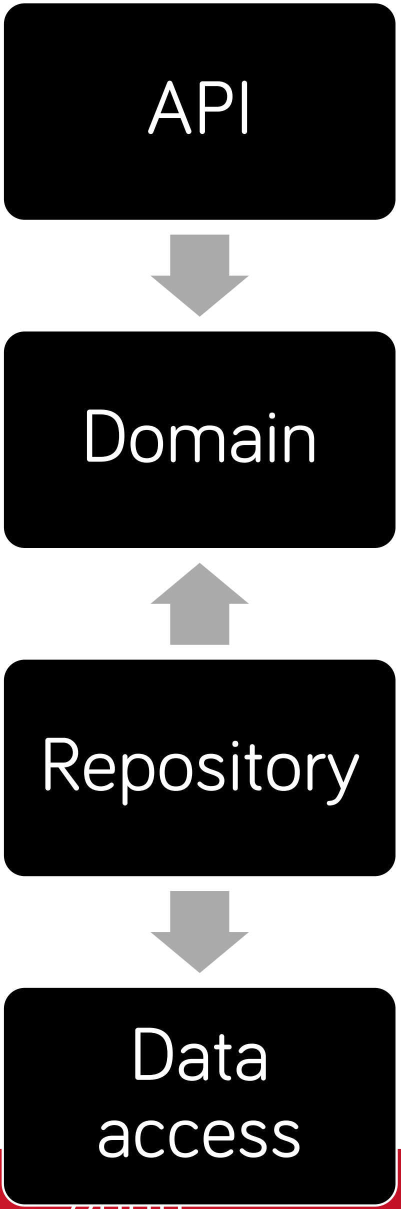
Repository pattern

Data access abstraction: repository pattern



Kép forrása: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10)

Repository sample



```
public User GetUserById(int userId)
```

```
public class User
{
    public int UserId;
    public string FirstName;
    public string LastName;
}
```

```
public class UserRepository : IUserRepository
{
    public User GetUserById(int userId)
    {
        //DB Call here…
    }
}
```

Abstracting data access

Repository: class

- Multiple repositories
  - Per entity or entity group
  - Product and category; Orders; …
- One repository: one class
  - It is typically used with an interface
    - Testing, mocking (see later)
  - CRUD operations
  - Works with business entities
- All database-specific details are enclosed here
  - Technology (EF Core) and platform dependent (MS SQL)

Repository sample: CRUD

```
class ProductRepository : IProductRepo
{
   List<Product> List() { ... }
   Product FindById(int Id) { ... }
   void Add(Product entity) { ... }
   void Delete(Product entity) { ... }
   void Update(Product entity) { ... }
}
```

Repository sample: operations

```
class ProductRepository : IProductRepo
{
   void AddProductToCategory(
      Product p, Category c){...}

   void StopSellingProduct(
      Product p){...}
}
```

Repository Pattern properties

- Abstraction above the data layer
- A specific location where the way data is accessed can be changed
- Location of datatables
- It can easily be changed to another implementation
- It hides the details
- Several alternative implementations

Repository Pattern

- Business logic independent of data layer
- Separation of domain logic and data access
- Data access/database replacement
- Unified management of multiple data sources
- Change of data storage paradigm (e.g. nosql, lucene, etc.)
- Support for unit tests
- Better parallelization of development
- Reduction of duplicate queries
- Better control over queries
- Object oriented API
- Uniform rules
- Caching

- Another layer - more work
- It reduces the power of ORM technologies
- Risk: Data storage API still leaks into business logic

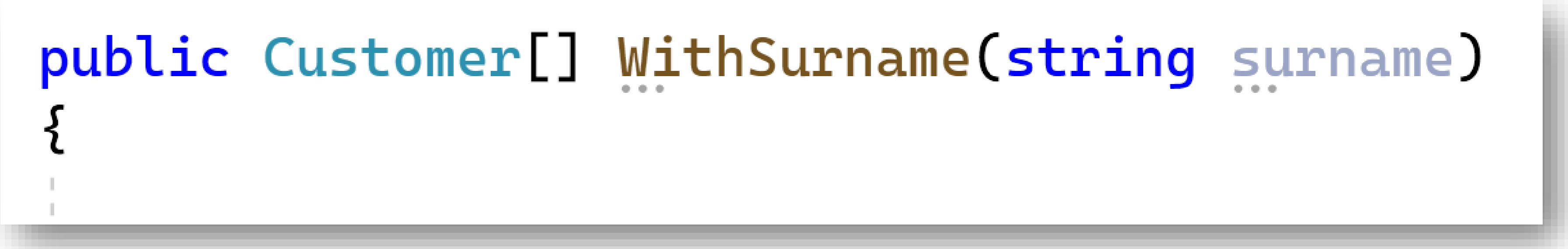
ORM and the repository

- Different goals
- Repository
  - Abstraction and consolidation of all functions related to storage
  - Architectural pattern
- ORM
  - Abstraction of supported relational database access
  - Used behind Repository as an "implementation detail"

Repository functions in practice

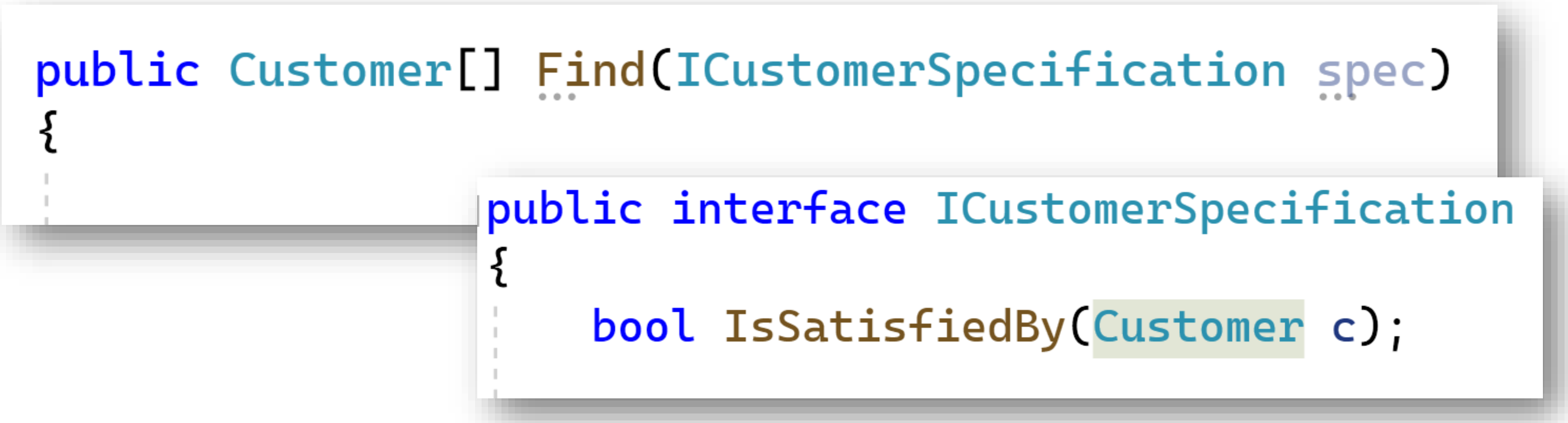
- Add and remove entities
- The interface uses tables and collections
- Transaction management/closing is not here
- **Queries according to the desired criteria**

Query functions



```
public Customer[] WithSurname(string surname)
{
```

Generalized queries

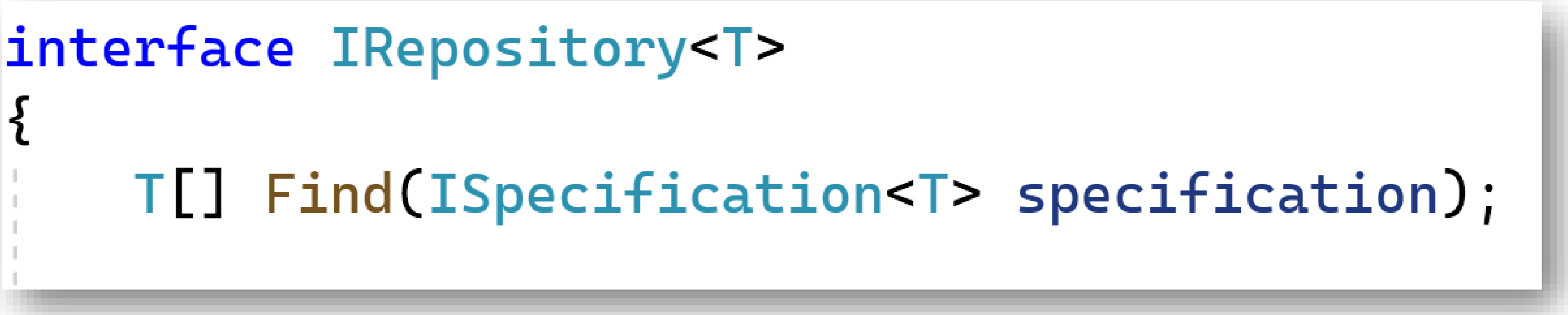


```
public Customer[] Find(ICustomerSpecification spec)
{
```

```
public interface ICustomerSpecification
{
    bool IsSatisfiedBy(Customer c);
```

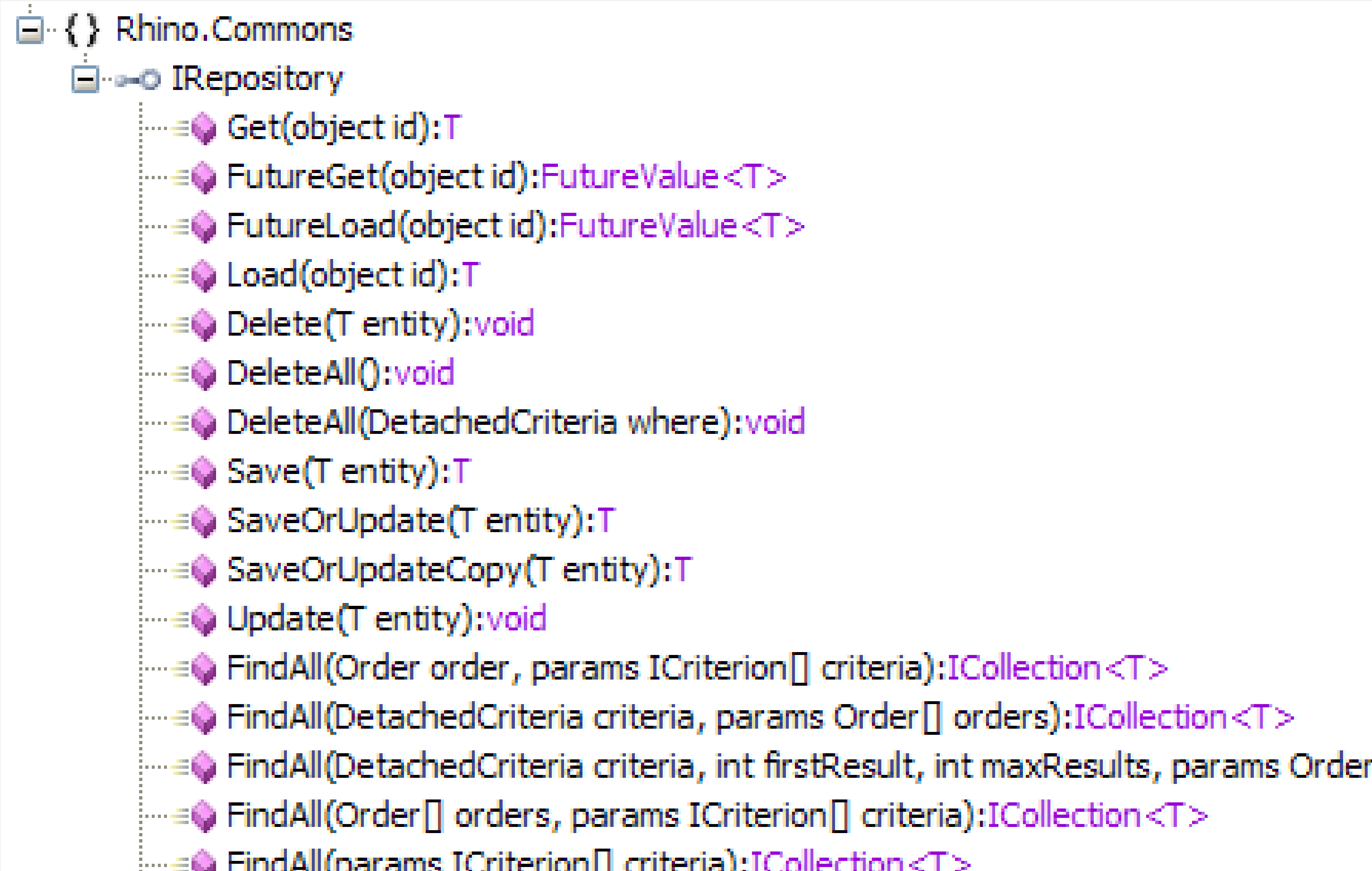
- Part of the domain model
- Composable (complex expressions)
- Must work in-memory or generating SQL

Generic repository pattern



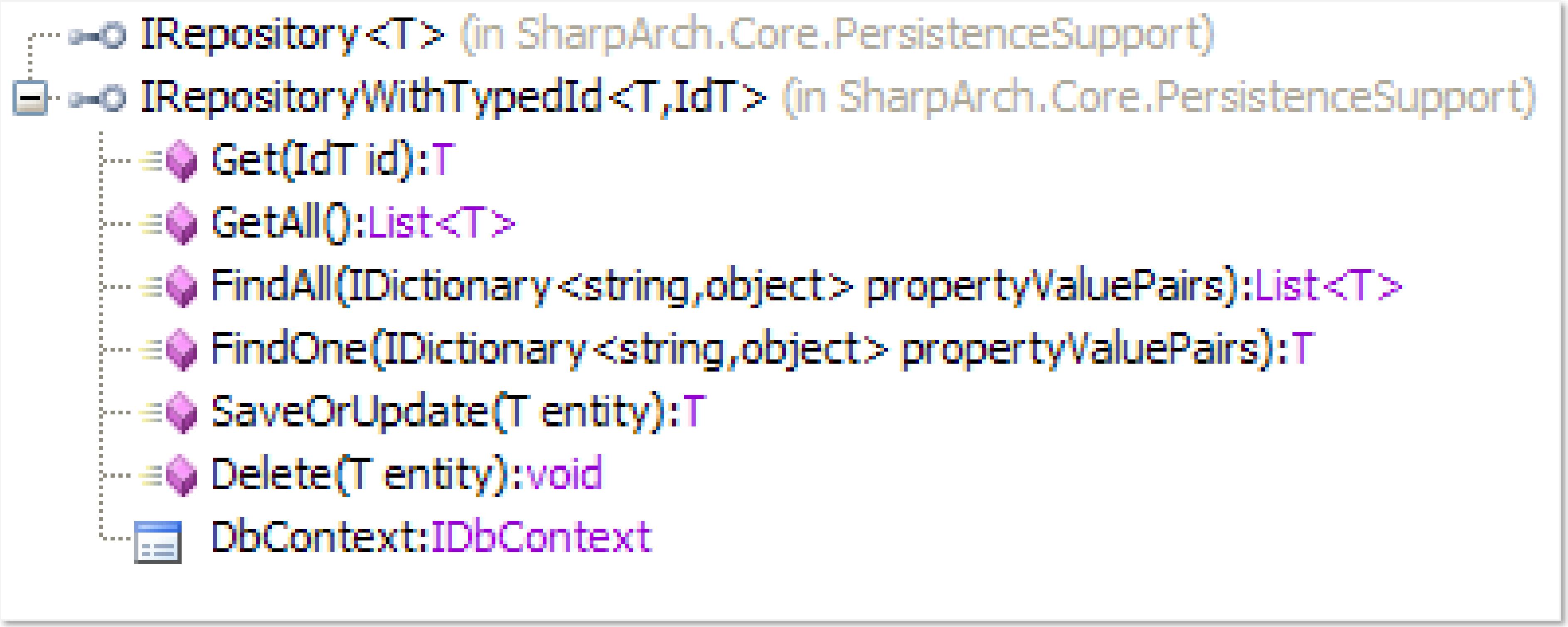
```
interface IRepository<T>
{
    T[] Find(ISpecification<T> specification);
```

Rhino Commons



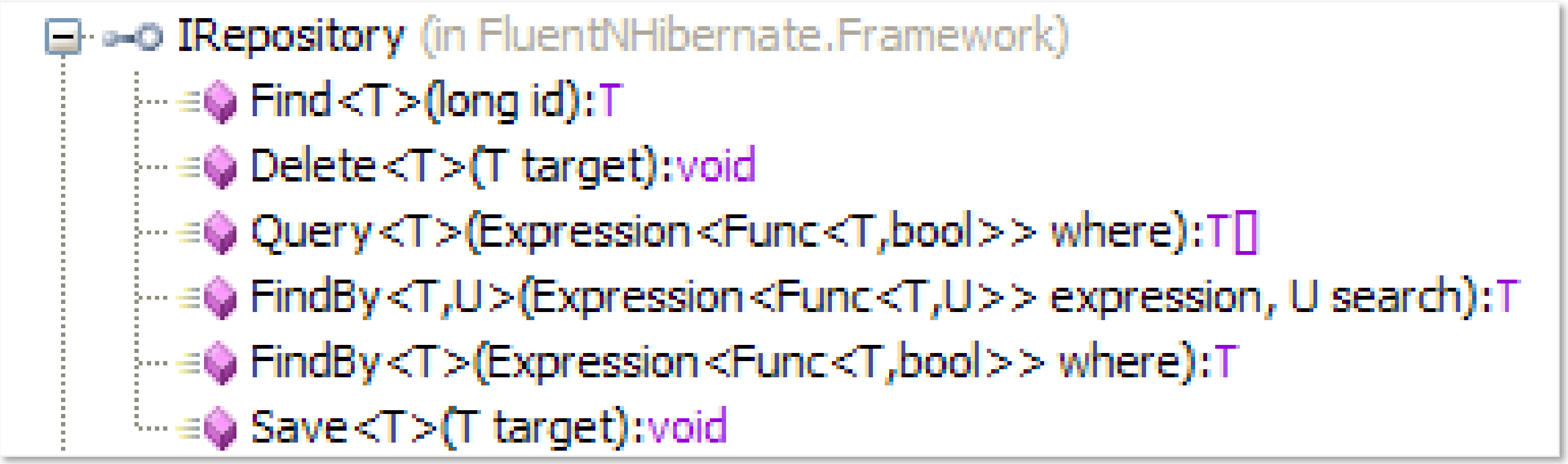
```
{ } Rhino.Commons
    IRepository
        Get(object id):T
        FutureGet(object id):FutureValue<T>
        FutureLoad(object id):FutureValue<T>
        Load(object id):T
        Delete(T entity):void
        DeleteAll():void
        DeleteAll(DetachedCriteria where):void
        Save(T entity):T
        SaveOrUpdate(T entity):T
        SaveOrUpdateCopy(T entity):T
        Update(T entity):void
        FindAll(Order order, params ICriterion[] criteria):ICollection<T>
        FindAll(DetachedCriteria criteria, params Order[] orders):ICollection<T>
        FindAll(DetachedCriteria criteria, int firstResult, int maxResults, params Order
        FindAll(Order[] orders, params ICriterion[] criteria):ICollection<T>
        FindAll(params ICriterion[] criteria):ICollection<T>
```

Sharp Architecture



```
IRepository<T> (in SharpArch.Core.PersistenceSupport)
IRepositoryWithTypedId<T,IdT> (in SharpArch.Core.PersistenceSupport)
    Get(IdT id):T
    GetAll():List<T>
    FindAll(IDictionary<string,object> propertyValuePairs):List<T>
    FindOne(IDictionary<string,object> propertyValuePairs):T
    SaveOrUpdate(T entity):T
    Delete(T entity):void
    DbContext:IDbContext
```

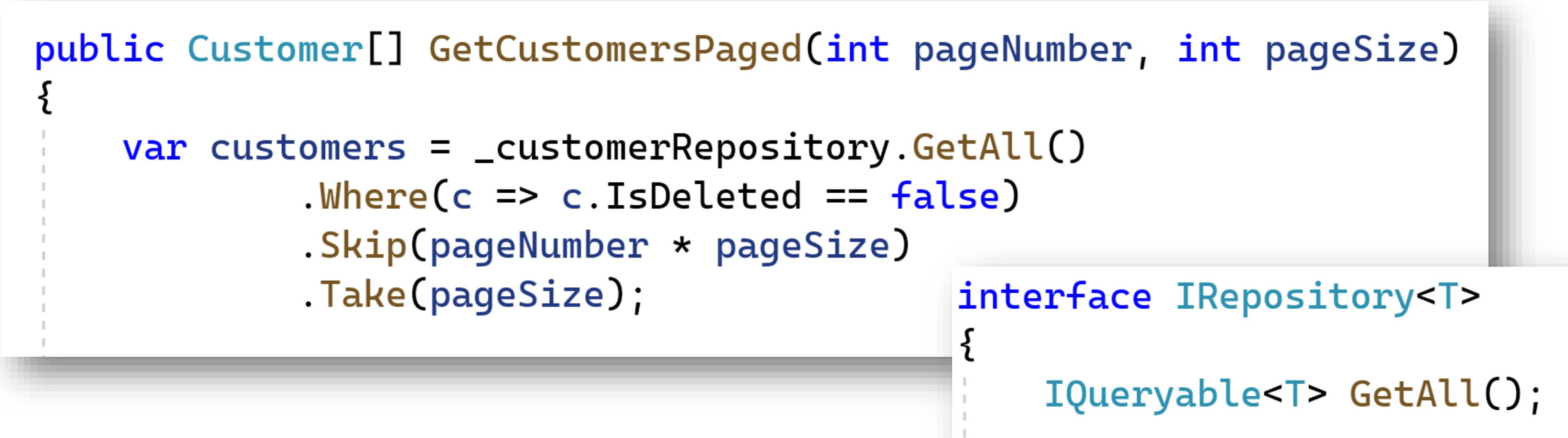
Fluent NHibernate



```
IRepository (in FluentNHibernate.Framework)
    Find<T>(long id):T
    Delete<T>(T target):void
    Query<T>(Expression<Func<T,bool>> where):T[]
    FindBy<T,U>(Expression<Func<T,U>> expression, U search):T
    FindBy<T>(Expression<Func<T,bool>> where):T
    Save<T>(T target):void
```

Is IQueryable usable here?

Using **IQueryable<T>** GetAll<T>() method in **BLL**:



```
public Customer[] GetCustomersPaged(int pageNumber, int pageSize)
{
    var customers = _customerRepository.GetAll()
            .Where(c => c.IsDeleted == false)
            .Skip(pageNumber * pageSize)
            .Take(pageSize);
```

```
interface IRepository<T>
{
    IQueryable<T> GetAll();
```

- The SQL executed outside the repository!
- Query technology can also infiltrate domain areas of the application ☹
- Called „Leaky abstraction”

About transactions

- It is not the responsibility of the repository
- Unit of Work pattern
  - NHibernate session
  - EntityFramework DataContext
- Establishing transaction boundaries is the responsibility of business logic

Criticism of the repository pattern

- "**Redundant**": no additional abstraction layer is needed
- It **complicates** the code, since "every" query must be written explicitly, LINQ cannot be used
- If we use IQueryable, we lose the benefits of the pattern because the query is ultimately generated by the BLL and we are tied to IQueryable
- "Return" to the world of stored procedures
- **Testing**: generally not easier to mock

Advantages of the repository pattern

- **Domain-driven:** provides a domain-specific view of the database - as opposed to queries formulated in LINQ, which are general
- **Hide/Aggregation**: hides the complexity of related entities (/tables) and gives a clearer view
- **Testing**: it's easier to mock for a small project

Guideline

- What is the goal?
  - Testing, maintenance, responsibilities, ...
  - Don't get caught up in the names (BLL, DAL, repo, …)!
- What is business logic? Where is?

1. In queries
   - With the repository pattern, the BLL can be almost "empty", the repository contains a lot of logic
   - Thus, the repo must be tested when we are testing!
2. Other logic, on data in-memory?
   - These codes should be separated and tested there!
   - They can get the data from the repo or EF...
3. Other validation logic?
   - You can relocate to separate classes, etc.

EF and Repository

- Let's refine the repository pattern as we saw earlier
  - Not just an ID-based query...
  - Any conditions, filters can be specified
  - Add group, join, query, …
  - Let's enable transactions on the interface
- But this is the EF itself!
  - A very generic query interface
  - The implementation is in the database provider
  - DbContext is just a UoW implementation
- In addition to EF, do not use a repository because it already is a repository implementation
- The repository pattern is good in itself, so otherwise use it (like in Java for example)!