Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Ralf H. Reussner

Topic 07

Patterns of Enterprise Application Architecture

DSIS – DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

dsis.kastel.kit.edu

Overview on Next Lectures

Content

- Challenges of Enterprise Applications
- Domain Logic Patterns
- Persistence Patterns

Learning goals: Students are able to

- characterize enterprise systems and decide which characteristics a given application has
- describe patterns of structuring domain logic, data source architectural patterns, and object-relational structural patterns.
- select an appropriate pattern for a given design problem and justify selection with respect to its advantages and disadvantages

Important to understand:

- there is no single solution that always fits
  - depends on system at hand, all solutions are different
  - ➔ different patterns solve same problem differently
- Patterns depend on each other
  - not all patterns fit well together
- patterns offer no complete solution
  - merely starting point
  - always adapt idea to the system at hand

[Fowler]

Enterprise Applications

- Enterprise application
  - software application that supports businesses, in particular
    - support of business processes
    - process business transactions and data
  - mainly about display, manipulation, and storage of data
  - specific challenges, different from other domains
  - ≠ enterprise architecture (= “organizing logic for business processes and IT infrastructure” [MIT CISR 2007])

➔ Specific patterns have been identified to help

- Examples
  - Payroll systems
  - Online shops
  - Patient record management
  - Shipping tracking
  - Leasing system
- Counter examples
  - telecommunication systems
  - word processors
  - operating systems
  - plant controllers

[Fowler]

Properties of EA

- Persistent data
  - data lives longer than applications and hardware
  - existing data needs to be integrated
- Large amount of data
  - usually millions of records
  - efficient access required
- Concurrent data access
  - Can be hundreds of people at a time
  - Concurrent access can cause errors (e.g. inconsistencies)

[Fowler]

Properties of EA (cont‘d)

- Numerous different user interface screens
  - both experienced and inexperienced users
- Interfaces to other systems
  - integration with other enterprise applications
  - different mechanisms: REST, SOAs, CORBA, batch files, …
- Conceptual dissonance
  - different people may understand the same term differently
- Complex business „illogic“
  - business rules and processes are complex, but not necessarily logical
  - often cannot be changed for development
  - ➔ *consequence*: Logic needs to change

[Fowler]

Large Variety of Systems

1. Web Shop
   - many users browse and buy
   - business logic may be simple and well known
     - shopping carts, orders, availability information, shipment, payment
   - presentation that is compatible with many user devices

SCALABILITY
PERFORMANCE

2. Leasing Management System
   - complicated business logic
     - calculates monthly bills, handles events such as early returns and late payments, validates leasing contracts
   - few concurrent users

COMPLEXITY
MAINTAINABILITY

3. Expense Tracking for small company
   - few users, simple business logic
   - tight time-to-market and cost constraints
   - needs to be extensible
     - potential upcoming features, e.g. automatically make reimbursement, update tax issues, add reporting, connect to airline reservation systems, ... [Fowler, p.5]

TIME TO MARKET
EXTENSIBILITY

Layers of EA

Presentation (Front End)

- Handle interactions between user and software
- Display information
- Interpret commands

Domain (Middle, Business)

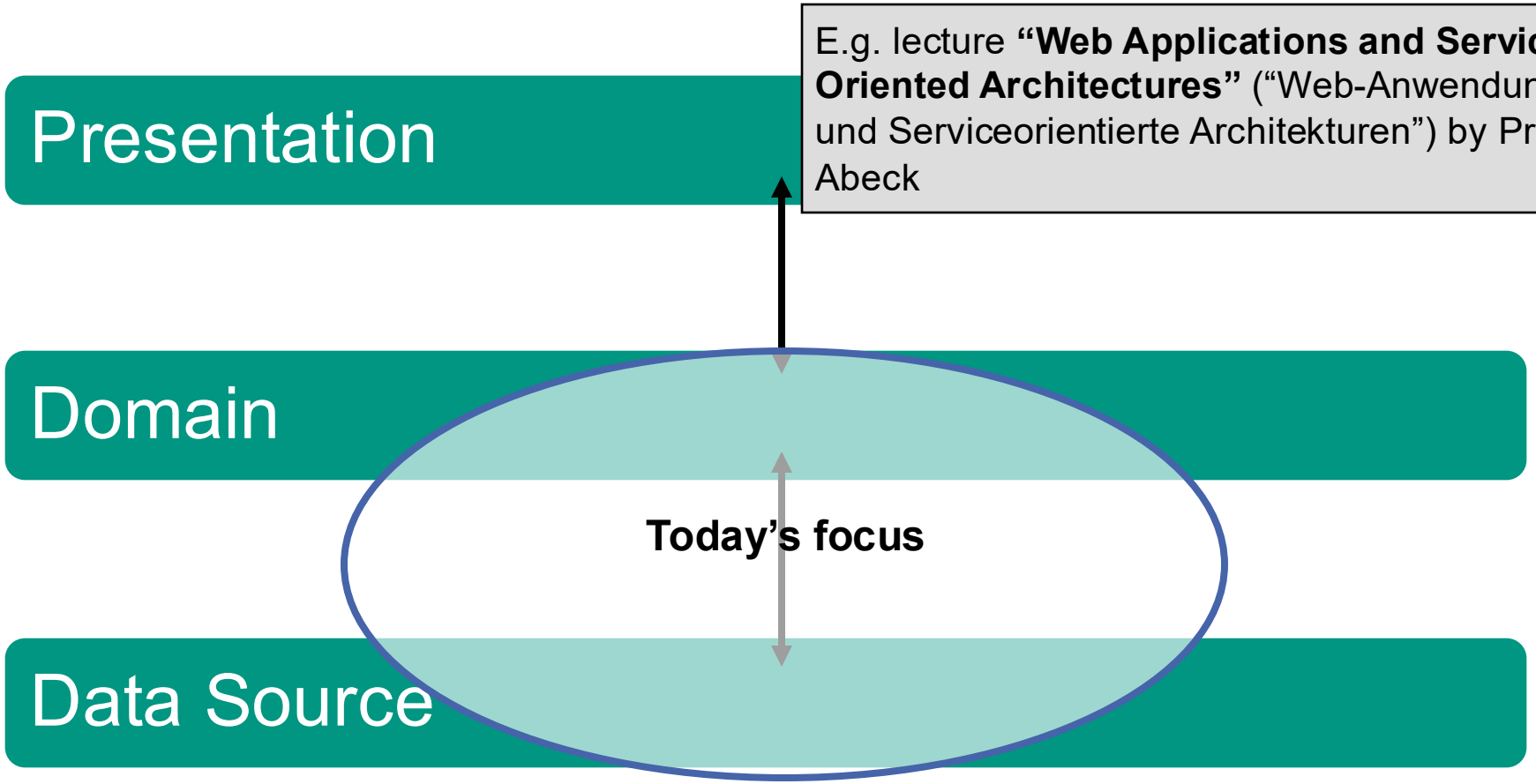
- Work that application needs to do in the domain
- Calculations on data, what to load and store

Data Source

- Communicating with other collaborating systems
- Example: *Database*

[Fowler]

Layers of EA



E.g. lecture **"Web Applications and Service-Oriented Architectures"** ("Web-Anwendungen und Serviceorientierte Architekturen") by Prof. Abeck

[Fowler]

PATTERN FAMILIES

Patterns of EA

- Patterns to solve the challenges of EA

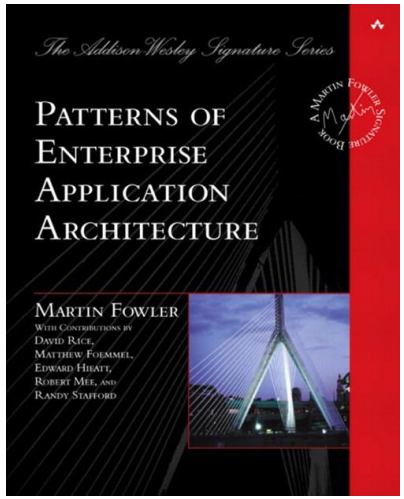
- Specific to problems in one of the layers
- Interactions with other patterns (from other layers)

- Pattern families:
  - Different patterns to approach one problem
  - Overview on the next slides

Pattern Families Addressed Today

- Domain Logic
  How to represent the business rules?
- Data source architecture
  How to separate domain logic and data source?
- Object-relational structural patterns
  How to map objects to relational databases?

- Online resource with short descriptions:
  http://www.martinfowler.com/eaaCatalog/index.html
- Full description: [Fowler], Patterns of EA Architecture



Other Pattern Families in [Fowler]

- Object-relational behavioural patterns
  How to organize the loading and storing?
- Object-relational metadata mapping patterns
  How to use metadata to generate mappings?
- Web presentation patterns
  How to organize different views on the application?
- Distribution patterns
  How to (not) distribute your application?
- Offline concurrency patterns
  How to ensure data integrity for long transactions?
- Session State patterns
  How to handle the state of transactions?

Just an overview, no need to memorize

Pattern Families

- Each family addresses one problem of EAs
- Contains different patterns how to solve it

Choosing a pattern

- Depending on requirements of system at hand
- Consider advantages and disadvantages of pattern
- Choice is not isolated
- Relation to other patterns for other problems

Patterns Are Only Starting Point

- Patterns rooted in practice
- Describe what architects often successfully do

However:

- Abstract from all the problems solved by them
- Half-baked solution
- Several possible solutions

- Only a starting point to think and decide

- Here EA patterns play the role of a style
  - Should not be mixed among themselves for the problem solved
  - (Of course, other patterns can still be used for other problems)

[Fowler, p.13]

Structure in the Following

- Presentation of the separate pattern families

- Addressed challenge and example
- Some patterns applied to that example
  - How it works
  - Advantages / Disadvantages
- When to use what

DOMAIN LOGIC PATTERNS

[Fowler, p.25-30, p.109-129]

Adressed Challenge

- Domain logic: Representation of business logic
- Possible challenges
  - High complexity of the domain logic
    Validation, Calculation, Concepts?
  - Need to be changeable
  - Need to connect with presentation and data source
- Choice depends on system at hand

[Fowler]

Example

- Recognize revenues for a contract
- A contract is the selling of a product
- For each product, the revenue is calculated differently

- So
  - Determine the product
  - Apply the correct algorithm to calculate revenue
  - Create a result object

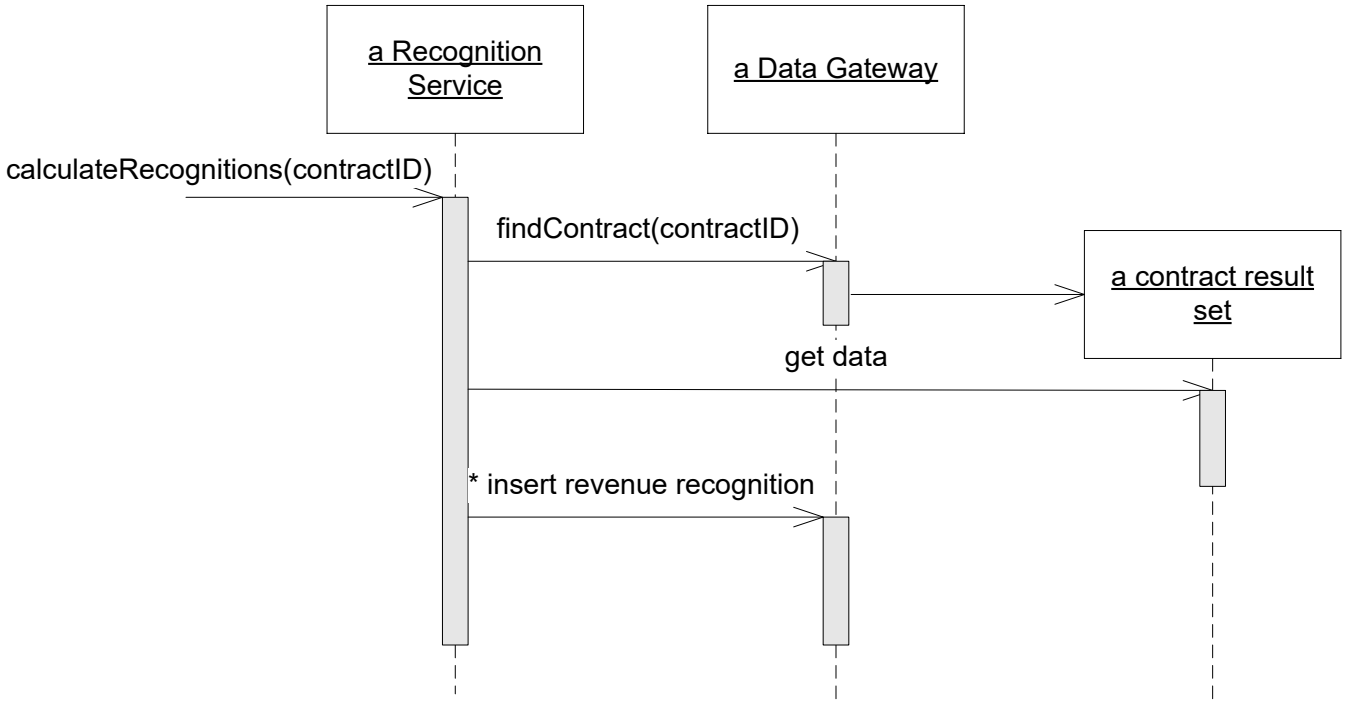
[Fowler, p. 112]

“Organizes business logic by procedures where each procedure handles a single request from the presentation.”

TRANSACTION SCRIPT

[Fowler]

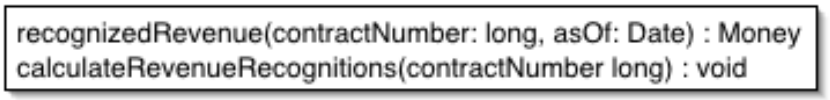
Transaction Script (pp. 110)



- Put all logic for this transaction in a procedure
- Retrieve contracts from data source, calculate, and store results

[Fowler]

Transaction Script



- Single procedure per transaction type
- Factor common behaviour out into subroutines

Advantages

- Simple procedures that developers understand
- Easy to connect to simple data sources
- Transaction boundaries are easy to determine

Problems

- Does not scale well with complex logic
- Tends to have duplicate code

[Fowler]

“An object model of the domain that incorporates both behavior and data”

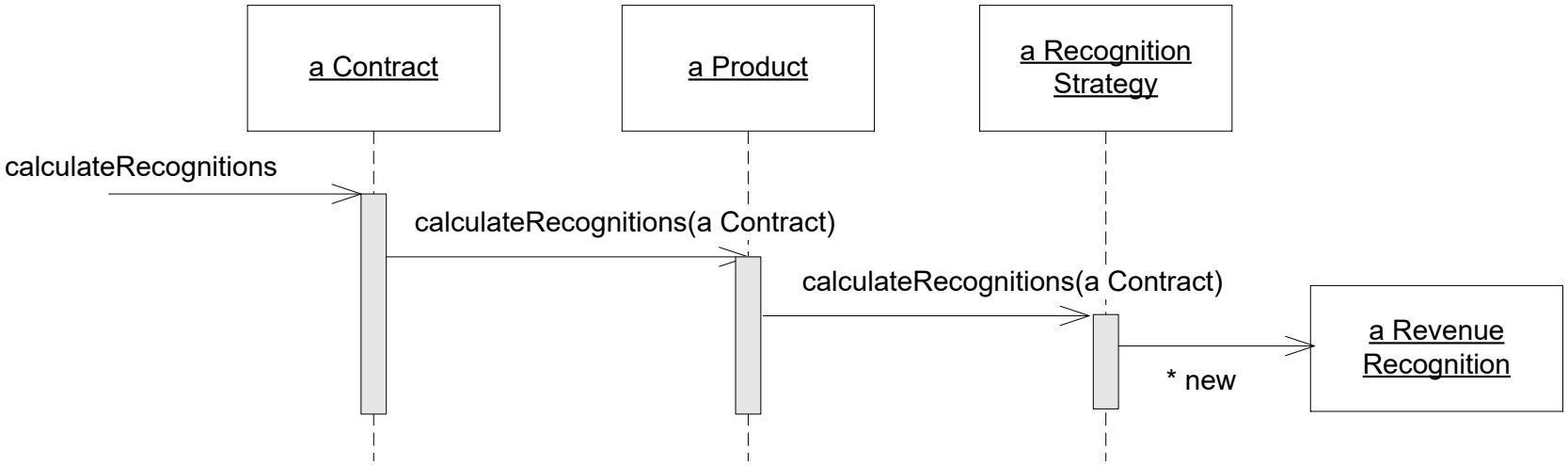
DOMAIN MODEL

[Fowler]

Domain Model?

- Domain Model
  - Artefact in the Unified Process
- Domain Layer
  - The layer with business logic in your application
- Domain Model Pattern
  - Designing the domain layer using a domain model
  - Low representational gap between design classes and domain model

Domain Model

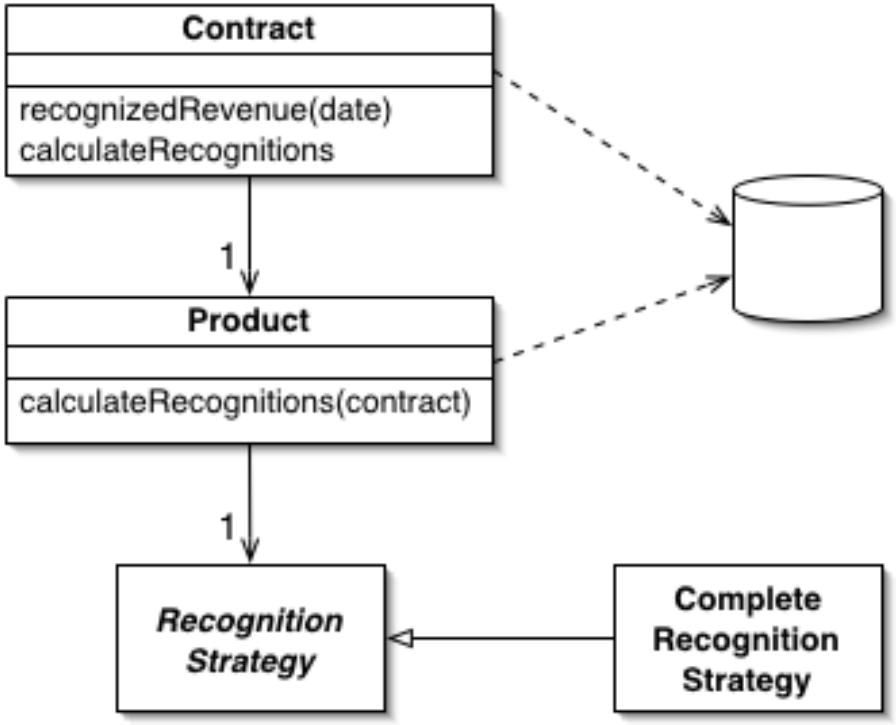


- Object-oriented thinking
- Identify concepts, and place logic with them
- Objects collaborate to do the transaction

[Fowler]

Domain Model

- Logic is kept with the concept
- Set of objects collaborate



Advantages

- Better organizes complex domain logic

Problems

- Steeper learning curve if not familiar with OO
- Mapping to data source more complex

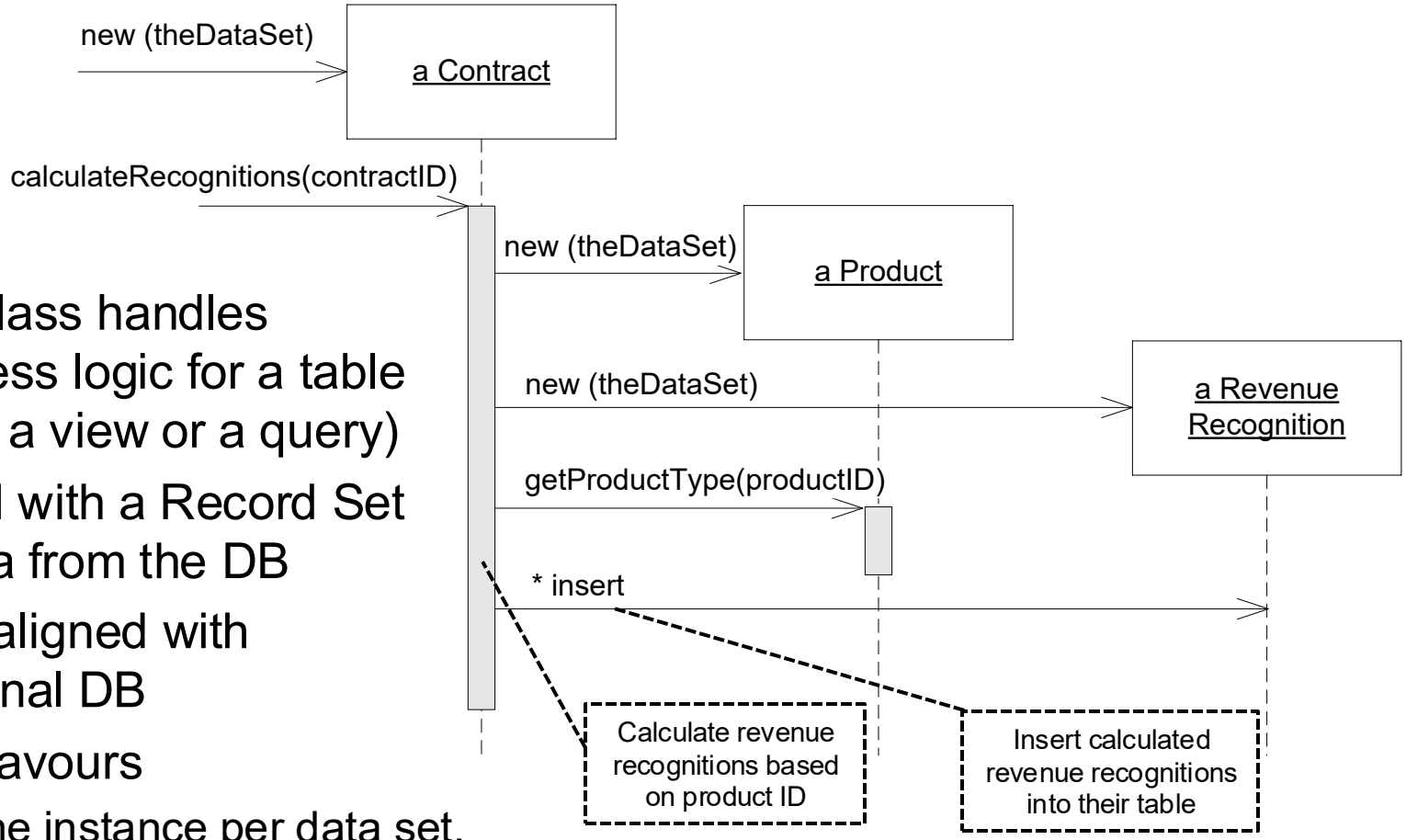
[Fowler]

“A single instance that handles the business logic for all rows in a database table or view”

TABLE MODULE

[Fowler]

Table Module

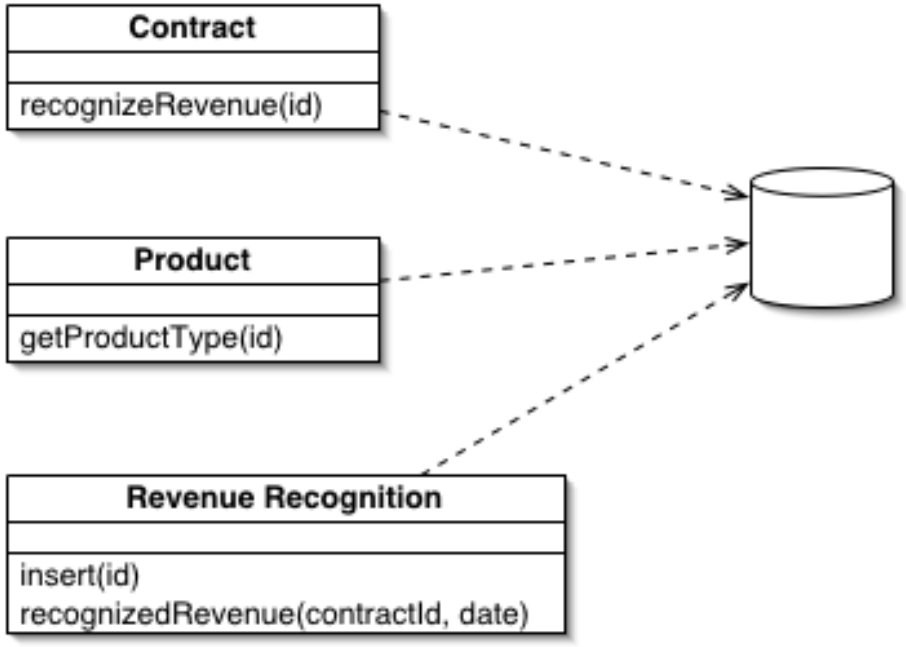


- One class handles business logic for a table (or for a view or a query)
- Called with a Record Set of data from the DB
- More aligned with relational DB
- Two flavours
  - One instance per data set, e.g., *per result of query (as shown in diagram)*
  - Static methods

[Fowler]

Table Module

- Objects do **not** have identity
- Works with table-like data structure



Advantages

- Straightforward mapping to data
- Separates logic for different concepts
- Useful if used technology supports it (COM, .NET)

Problems

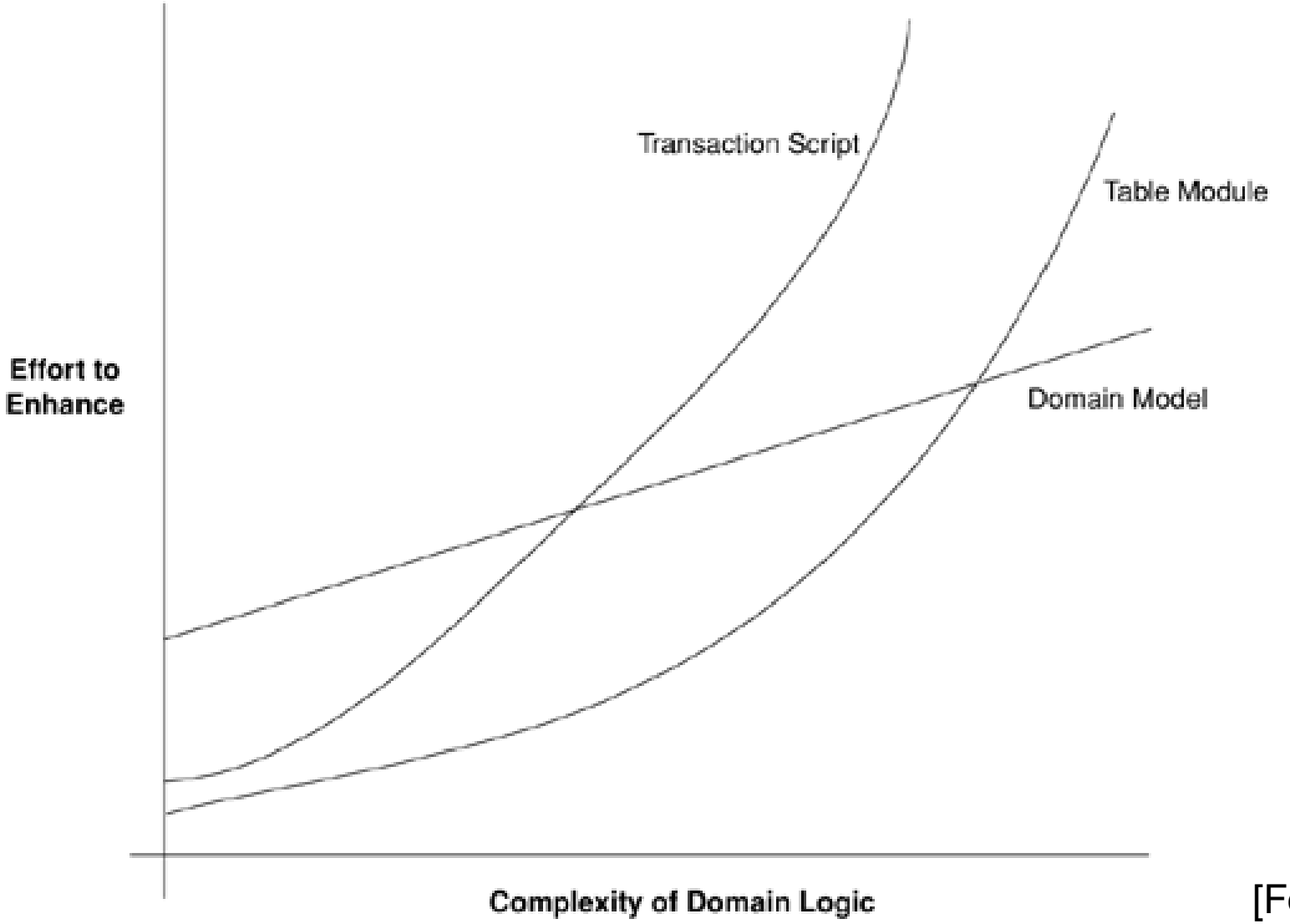
- No object instances: can be bad for complex logic

[Fowler]

When to use what

- Strongest factor: How complex is the domain logic?
  - See graph on next slide

Simplified View on Complexity and Effort



[Fowler, p.29]

When to use what

- Strongest factor: How complex is domain logic?
  - See graph on previous slide
- How complex is the mapping to data sources?
  - What choices on data source architectural level?
- Are developers familiar with domain models?
  - If yes, less disadvantages, more attractive
  - Domain model must be carefully designed and adhered to be successful.
  - If no, transaction script is easier to use and to maintain.
- What tools do you use?
  - Development environments / tools may favour a pattern
- Possible to combine all three

[Fowler p.29-30]

Example Systems

- No single architecture that fits all three example systems from slide 6
  - different non-functional requirements

➔ Identify specific problems
  - choose appropriate architectural design

- How do patterns relate here?
  - patterns are about choices and alternatives
  - evaluate which patterns fits your system‘s need
  - adjust pattern to your situation

[Fowler]

- You may want to use for the business logic of –
  - *Web Shop:* _____
  - *Leasing System:* _____
  - *Expense Tracking System:* _____

Example Systems

- No single architecture that fits all three example systems from slide 6
  - different non-functional requirements

➔ Identify specific problems
  - choose appropriate architectural design

- How do patterns relate here?
  - patterns are about choices and alternatives
  - evaluate which patterns fits your system‘s need
  - adjust pattern to your situation

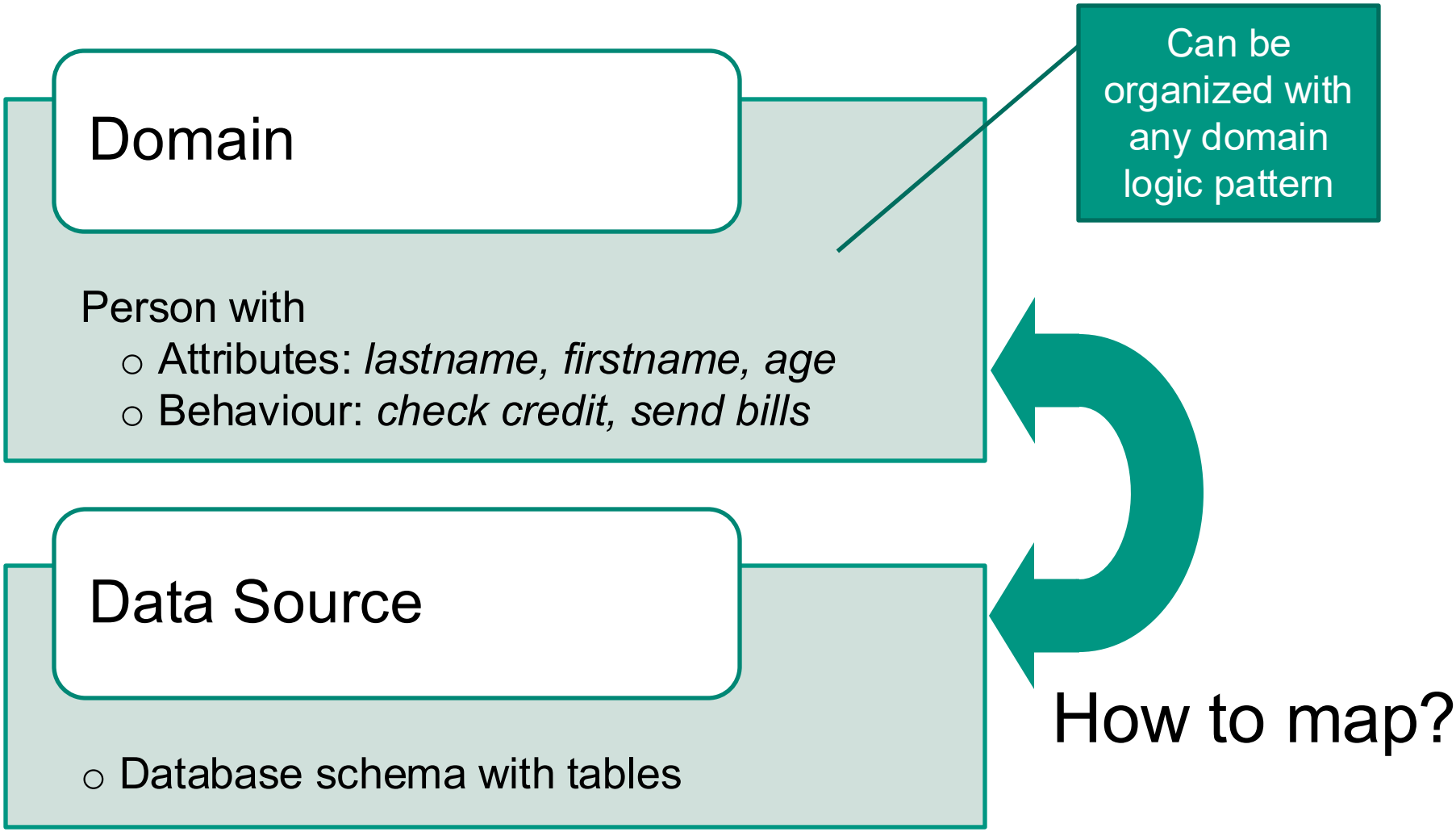
[Fowler]

- You may want to use for the business logic of –
  - *Web Shop:* Table Module (125) or Transaction Script (110)
  - *Leasing System:* Domain Model (116)
  - *Expense Tracking System:* Transaction Script (110) or Domain Model (116)

DATA SOURCE ARCHITECTURAL PATTERNS

[Fowler, p.33-38, p.143-179]

Motivating Example for Persistence



OO vs. DB

- Relational databases are the dominant paradigm for data storage today
  - but how do they fit together with object-oriented languages?
- Object-Orientation
  - objects with data and operations
  - references
  - aggregation & inheritance
- Relational Databases
  - flat tables (normalization!)
  - transactions (ACID)
  - relational algebra

Person
firstname lastname age

ID	Fname	Lname	Age
1	John	Doe	37
7	James	Bond	42

➔ Possible Solutions
- manual storage (files, JDBC)
- XML files or XML databases, OO databases etc.
- O/R mapping frameworks

Side Remark: „Database Thaw“

- Relational databases are not the only paradigm for data storage any more
- Rise of so-called NoSQL data stores
  - Focus on scalability by distribution
  - Often used together with relational databases – *„polyglott persistence“*
  - Trade offs: consistency versus latency, consistency vs availability
  - Often only provide *„eventual consistency“*
- Different types, e.g.
  - Key-value stores
  - Document stores
  - Graph stores
  - Column stores
- Still need to decide how to map objects to NoSQL data stores
  - Mapping solutions available, e.g. Hibernate OGM, Kundera
- More detailed discussion of underlying trade-offs by [Brewer 2012]

cf. https://martinfowler.com/articles/nosqlKeyPoints.html

Architectural Patterns for Data Access

- Architecture at the layer of data management
  - different ways of accessing a database
  - ➔ Goal: separate access to a relational database (SQL) from domain logic

- Specific patterns –
  - Basic pattern: Record Set (→ JDBC ResultSet)
  - Table Data Gateway
  - Active Record
  - Row Data Gateway
  - Data Mapper

- Another helper pattern: Identity Map

[Fowler]

Before going into architectural patterns,
first a basic pattern:

“An in-memory representation of tabular data”

RECORD SET

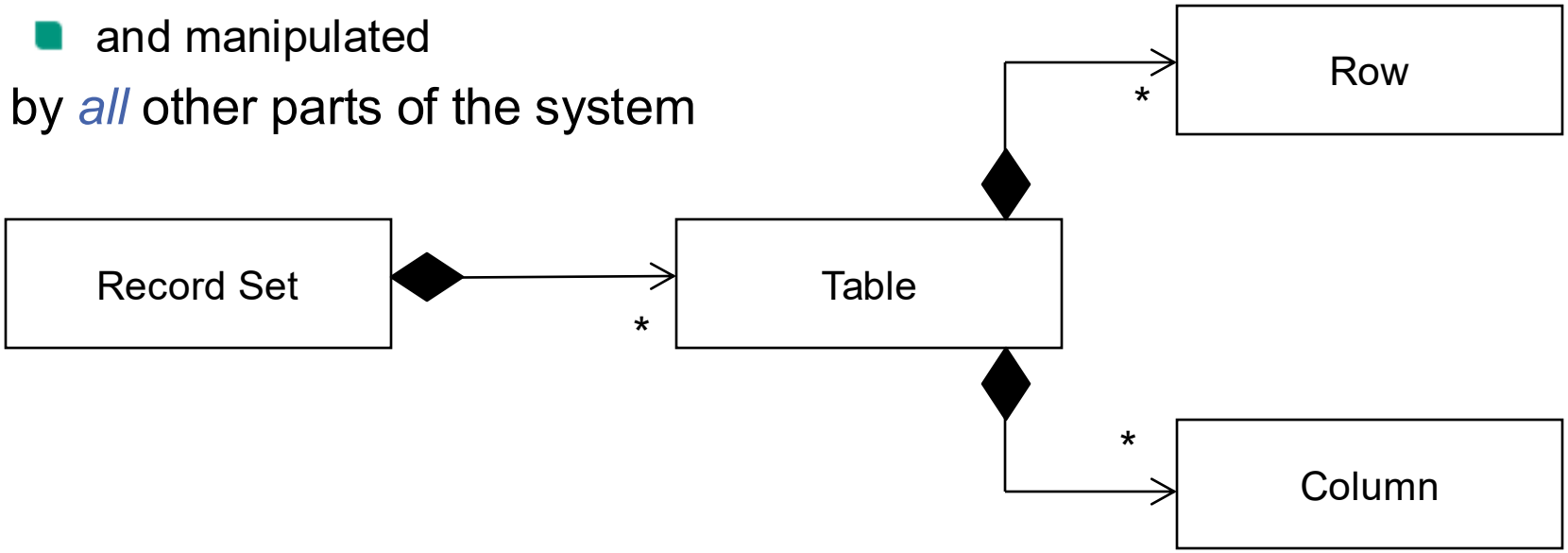
[Fowler, p.508-510]

Record Set (pp. 508)

"An in-memory representation of tabular data"

- In-memory object that looks like the result of an SQL query, but
  - can be easily generated
  - and manipulated

  by *all* other parts of the system



➔ *useful for simple client DB-server applications*

[Fowler, p.508]

Characteristics

- Usually not self-built but created by frameworks / platforms: e.g. Microsoft's ActiveX Data Objects (.NET) or JDBC result set
  - can be propagated through all tiers
- If equipped with serialization
  - employed for remote access
    - acts as a Data Transfer Object (DTO)
- Be careful: *Uses an implicit interface*
  - ADO.NET offers possibility for strongly typed Record Sets

```
ResultSet rs = stmt.executeQuery(sql);

While (rs.next()) {
   int id  = rs.getInt("id");
   String firstname = rs.getString("Fname");
   String lastname  = rs.getString("Lname");
   int age = rs.getInt(„age");
}
```

[Fowler]

➔ where to place DB access code?

[Fowler p.508-510]

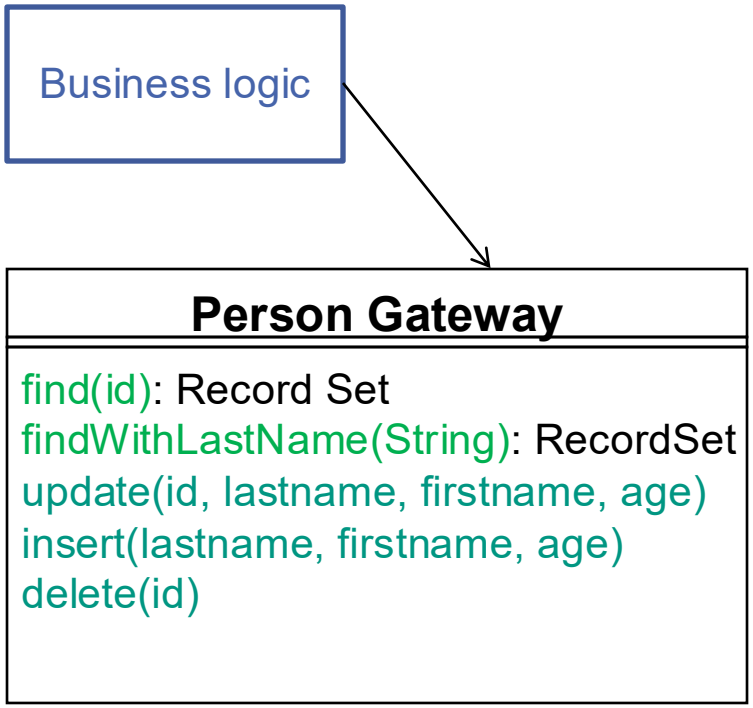
“An object [in fact a class] acts as a gateway to the database table. One instance handles all the rows in the table.”

TABLE DATA GATEWAY

Table Data Gateway (pp. 144)

"An object acts as a gateway to a database table. One instance handles all the rows in the table."

- Simple(st) Solution for DB access
- Aim: Separation of SQL statements
  - i.e. database access with queries and modifications, as well as
  - code that deals with that data
- Should cover: CRUD methods
  - each methods maps the input parameters to an SQL statement and executes the SQL command



➔ *not very useful for a Domain Model implementation*
rather for Transaction Script (but would nearly duplicate Table Module)

[Fowler]

Gateway or Façade?

Façade

"An object encapsulating oo dependencies of several other objects"

- Façade simplifies a more complex API
- Usually done by the writer of the service for general use

Gateway

"An object that encapsulates access to an external system or resource"

- Idea similar to Façade
- But: Gateway is written by client for its particular use

Proxy

"An object that acts as a substitute for another one"

- Signatures of offered methods identical to the ones of the substituted object (like "implements the same interface"
- Usually offered (often generated) by the writer of the object to be substituted to simplify access or enrich functionality (security, calls to remote deployments, …)

[Reussner / cf. Fowler p.468]

“An object that wraps one row in a database table or view, encapsulates the database access, and adds domain logic on that data.”

ACTIVE RECORD

[Fowler]

Active Record (pp. 160)

"An object that wraps one row in a database table or view, encapsulates the database access, and adds domain logic on that data"

- OO approach: Bring data and functionality together
  - provides an intuitive concept to represent data

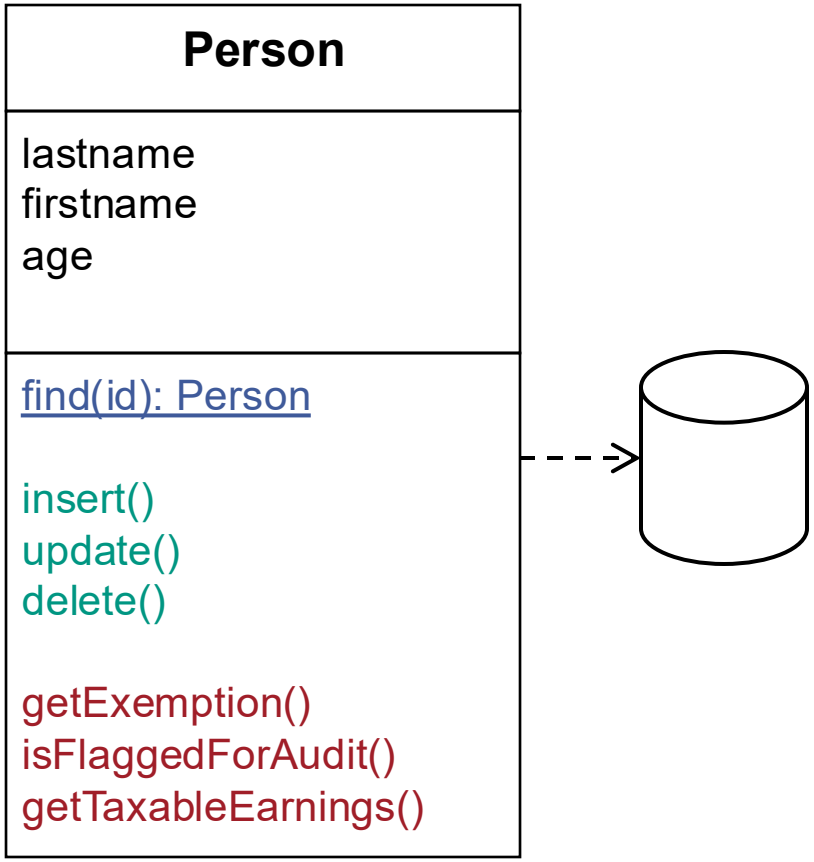
➔ Make all person objects know how to read and write their data from and to the database

[Fowler]

Solution / Example

- Each Active Record is responsible for saving and loading to the database
  - static finder methods wrap commonly used SQL queries and return Active Record objects
- Domain logic is in and acts on Active Records

➔ Active Record is a domain model whereas the classes match the database schema very closely
  - one field in the class for each column in the table



[Fowler]

When to use it

- Good choice for very simple domain logic
  - as it is easy to build and easy to understand then
- *Database schema and Active Record need to be isomorphic*
  - i.e. there needs to be a 1:1 mapping
- Not to use with complex business logic
  - with direct object relationships
  - and inheritance does **not** work well with Active Records
    - as this would end up with very messy code
    - use Data Mapper instead
- Diadvantage
  - Does not support separation of concerns (mixes domain logic and database access in one class)

[Fowler]

“An object that acts as a gateway to a single record in a data source. There is one instance per row.”

ROW DATA GATEWAY

[Fowler]

Row Data Gateway (pp. 152)

"An object that acts as a gateway to a single record in a data source. There is one instance per row."

Motivation

- putting database access code and in-memory-objects together (as in Active Record) increases complexity
  - as it mixes database access and business logic
- DB access code may need to cover several SQL variations for every database used
  - if no abstraction layer like JDBC is used

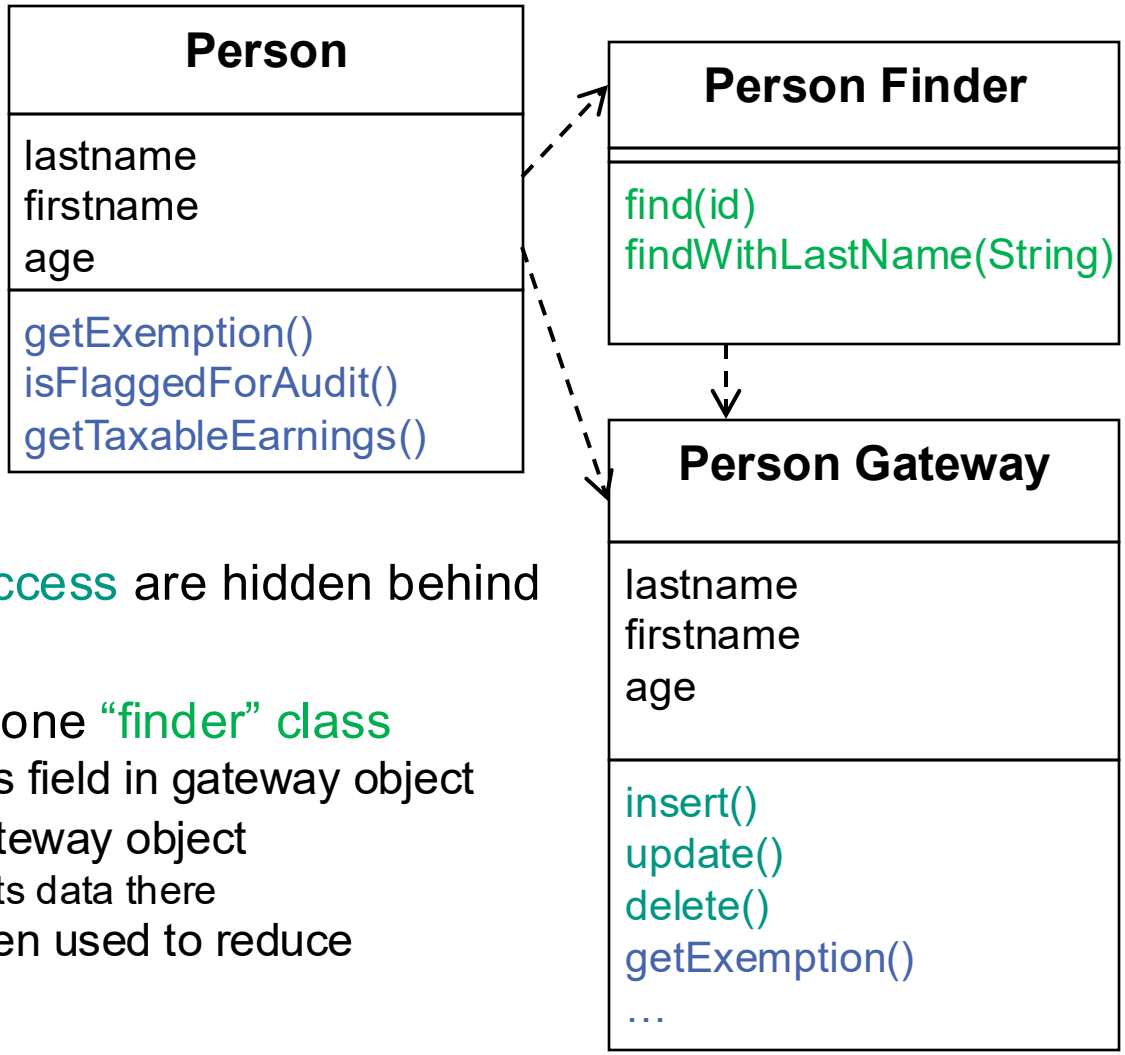
➔ In-memory objects are tied to DB

- makes testing slower and more complex because of DB accesses
- hard to refactor

[Fowler]

Solution / Example

- One object that looks like the database record structure, but
  - can be accessed with regular language mechanisms
  - is typed



- All details of data source access are hidden behind this interface
- Each DB table usually has one "finder" class
  - each DB column becomes field in gateway object
  - each DB row gets one gateway object
    - Person can even store its data there
  - Identity Map (pp. 195) often used to reduce DB lookups

[Fowler]

In Practice

- Row Data Gateways are good candidates for code generation
  - their code is generated using metadata as follows –
    - model data structure
    - generate database schema
      - Row Data Gateways are generated based on that model

➔ *The whole database access can be built automatically*

[Fowler p.153]

- Gateways shield domain layer from the database structure
  - changing database structure becomes possible
    - without changing the domain logic

  ➔ leads to (up to) three data representations:

  I. *(business logic (domain model))*
  II. *Row Data Gateway*
  III. *database*

  ➔ prefer Row Data Gateways that mirror the database structure

[Fowler p.152ff]

“Ensure that each object gets loaded only once
by keeping every loaded object in a map.
Look up objects using the map when referring to them.”

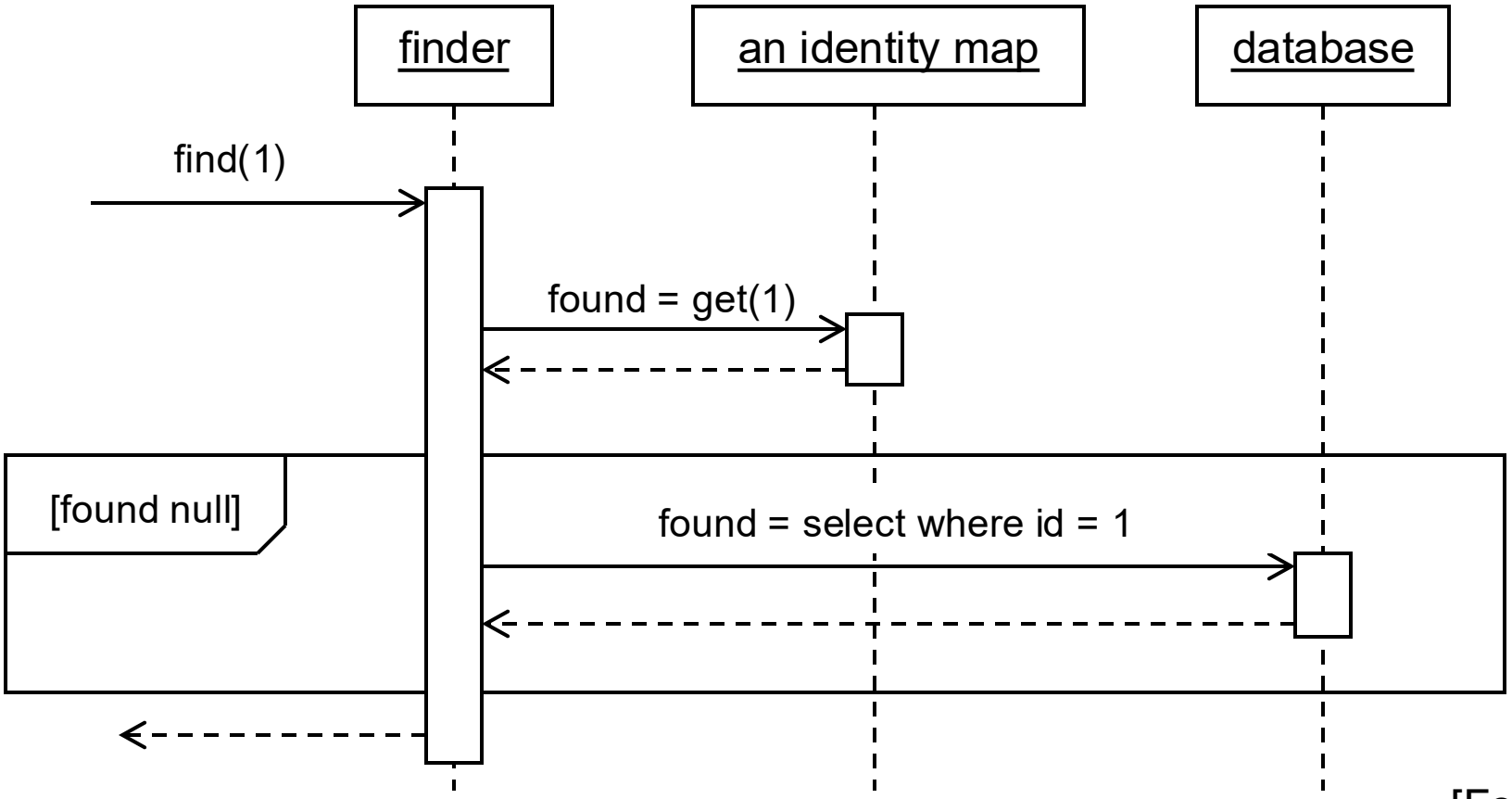
(required later)

IDENTITY MAP

[Fowler]

Identity Map (pp. 195)

Objects are retrieved from DB only *once*



[Fowler]

“A layer of Mappers that moves data between objects and database while keeping them independent of each other and the mapper itself.”

DATA MAPPER

[Fowler]

Data Mapper (pp. 165)

"A layer of Mappers that moves data between objects and database while keeping them independent of each other and the mapper itself."

- Remember: Objects and relational databases have different mechanisms for structuring data
  - Collections (fields)
  - Inheritance

  … are usually part of an object model

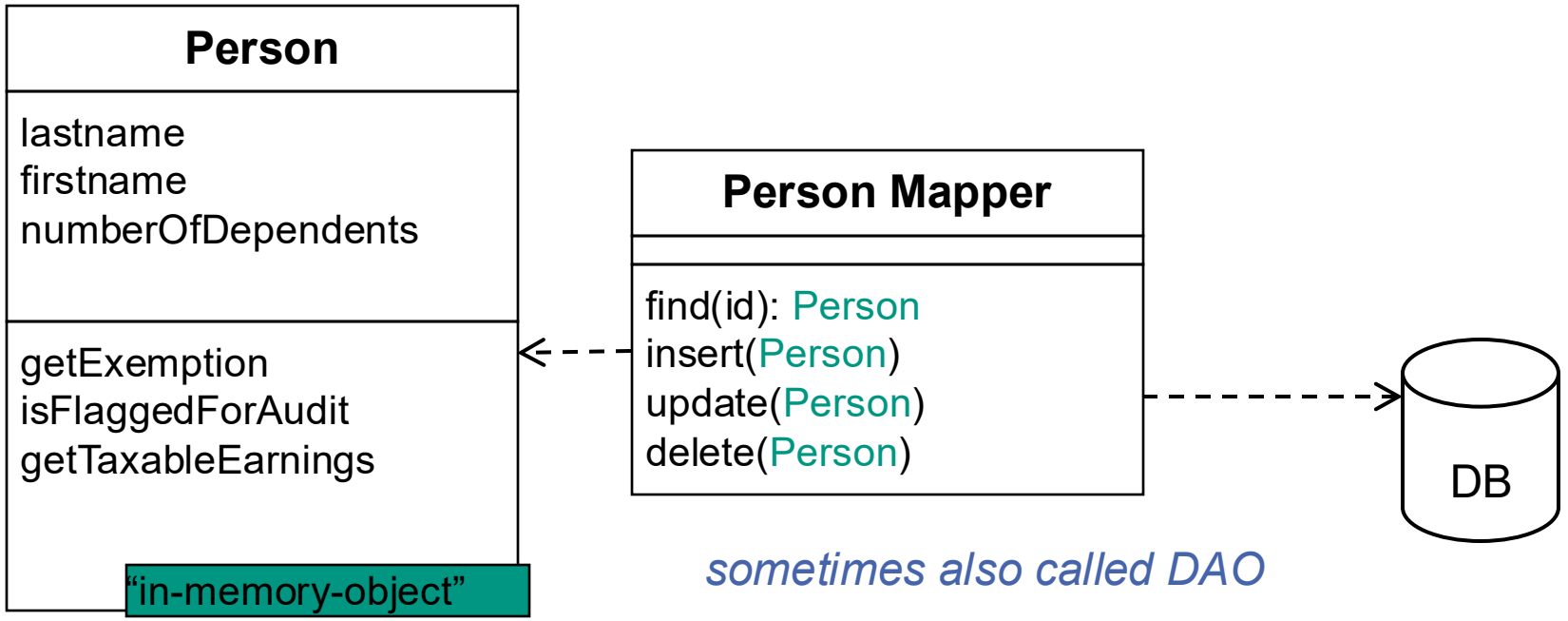
  ➔ not present in relational databases
- Leads to two schemas: object schema and relational schema that do not match up

➔ *Transfer between those two schemata may become complex*

[Fowler]

Solution / Example

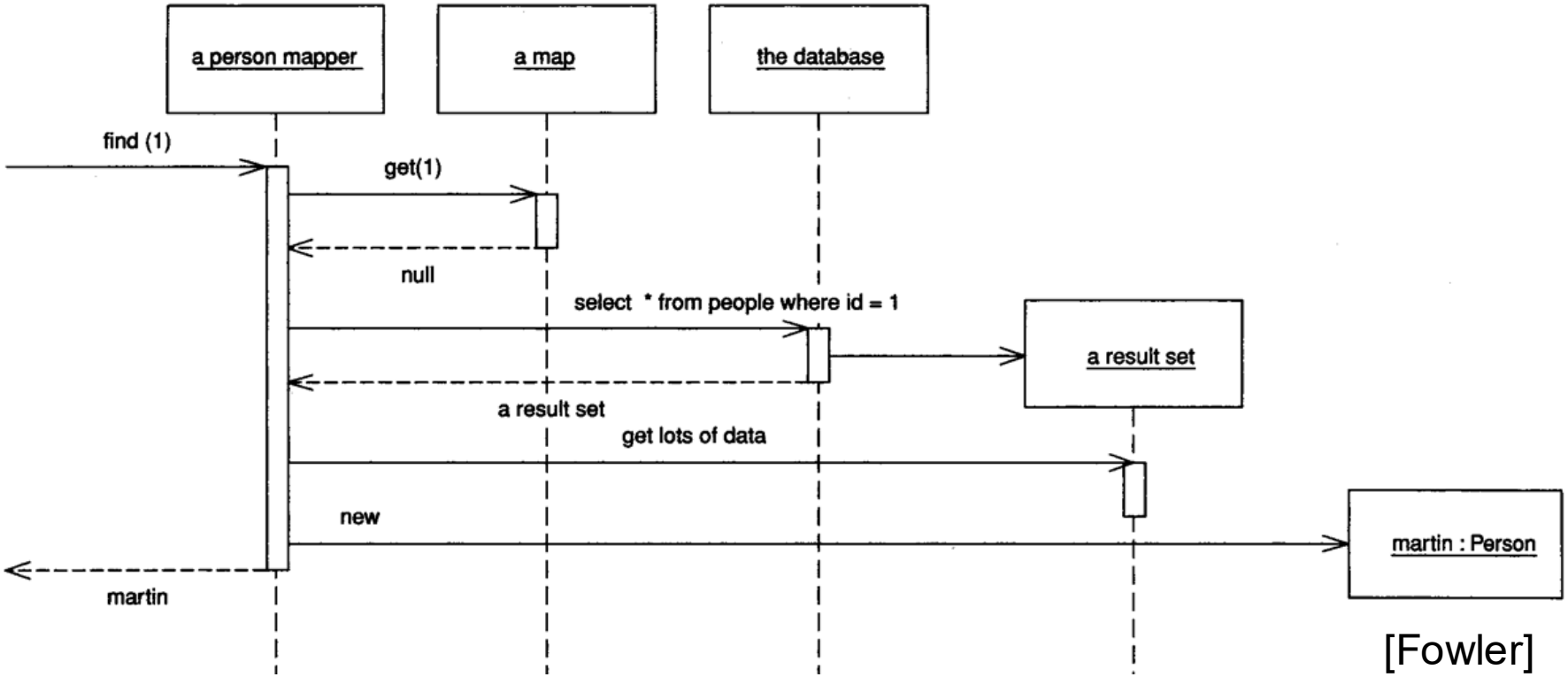
- A Data Mapper is a layer to separate in-memory-objects and database
  - transfers data between data source and in-memory-objects
    - maintains isolation between the two



[Fowler]

How it works

- In-memory-objects do not need to know about the database's presence
  - nor their SQL interface or schema
- To load a Person from the database, a client would call a find method on the mapper



[Fowler]

How it works (2)

- Mapping between domain objects and database columns can be done via explicit code
  - ➔ *i.e. for every domain object one data mapper needs to be created*

➔ ***Larman:*** ***"Don't try this at home!"*** *[p.622]*
"There is no need to create one yourself"

- Easier alternative: Metadata Mapping (p.306)
  - Metadata used to be held in separate class or file
    - today this can be done through annotations
  - ➔ Mappers may be created by code generation
    - or are provided by a mapping framework through reflective programming
  - Better buy an existing O/R mapper instead of writing an own "full-featured" one
    - "Full-featured" tends to get complicated: size, update and transaction support, concurrency, …

[Fowler p.170, p.38]

When to use it

- Database schema and object model usually evolve independently
- Complex business logic, where Active Record is not sufficient
  - but increases complexity by an extra layer
- Especially meaningful, if a domain model is used
  - MDD: Domain models are created and then transformed into database schema and object model
  - database schemas can be ignored then in design, build and testing
  - the whole translation between domain objects and database structure is handled by the mappers
- Legacy systems: Supports involving existing databases

[Fowler]

Discussion: When to use what?

Which data source architectural pattern would you choose for …?

- A **web shop** employing Transaction Script
  - _____
- A **leasing system** employing Domain Model
  - _____
- An **expense tracking system** employing Table Module
  - _____

Summary: When to use what?

Transaction Script

- Row Data Gateway: Explicit Interface, better to evolve
- Table Data Gateway: If Record Set framework

Domain Model

- Simple: Active Record
- Complex mappings: Data Mapper
- Gateways couple model too tightly to DB schema

Table Module (if Record Set framework)

- Table Data Gateway

Possible to combine all three

- E.g. Data Mapper with a Gateway to wrap tables that are considered external services [Fowler, p.36]

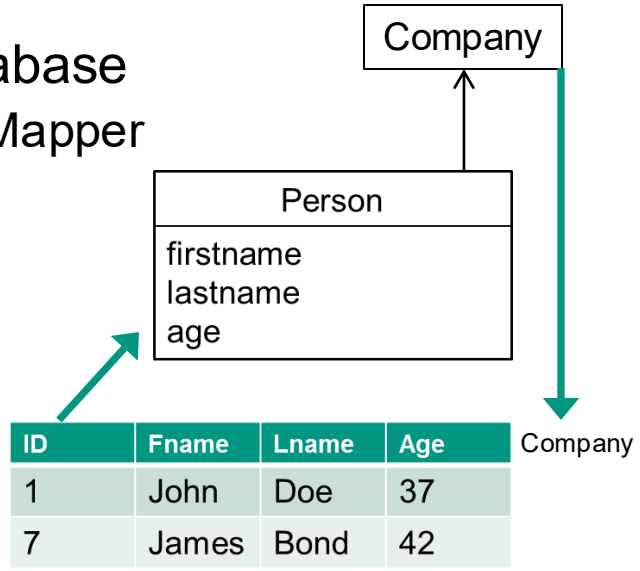
[Fowler]

OBJECT-RELATIONAL STRUCTURAL PATTERNS

[Fowler, p.45-47, p.278-296]

Object-Relational Structural Patterns

- Mapping of OO structures onto a relational database
  - especially when using a Domain Model & Data Mapper
- Identity and relational patterns
  - Not discussed in very much detail here
    - Example I: Identity Field (pp. 216)
      saves a database ID in an object to maintain identity between an in-memory object and a database row
    - Example II: Foreign Key Mapping (pp. 236)
      maps an association between object to a foreign key reference between tables
- Inheritance patterns
  - strategies for mapping inheritance
  - each with their own advantages and disadvantages



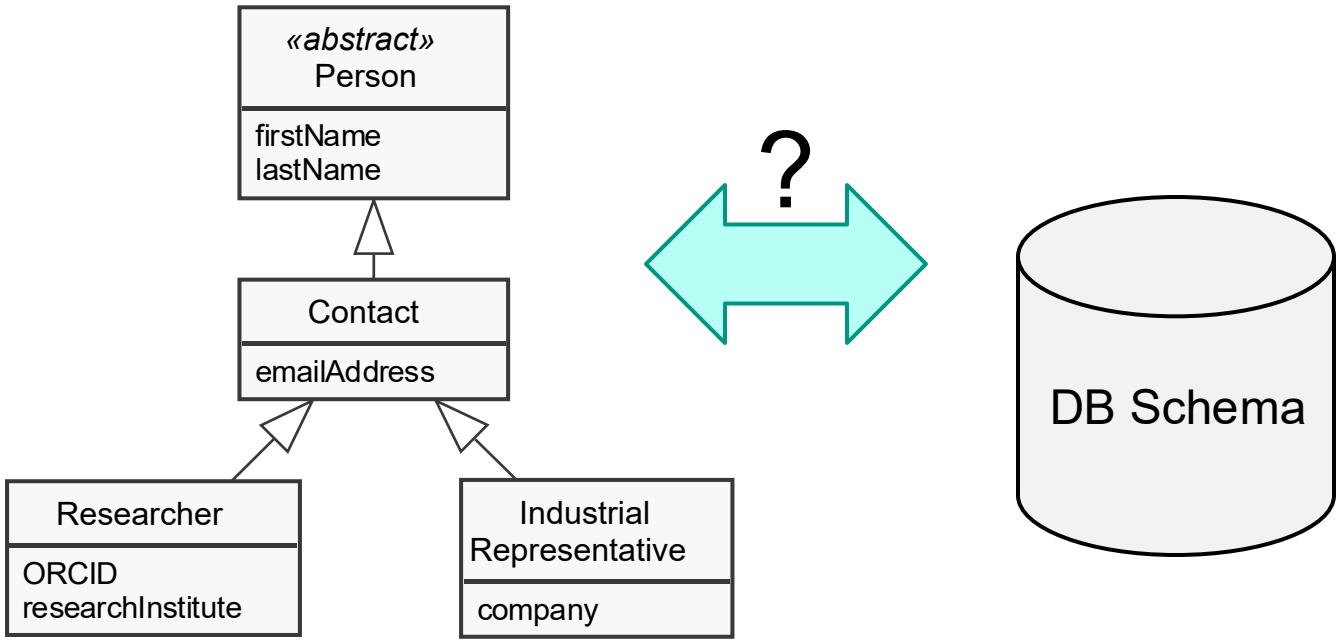
Motivating Example

- OO structure in domain logic

vs.

- Relational tables in data source

➔ How to map inheritance structures?



[Fowler]

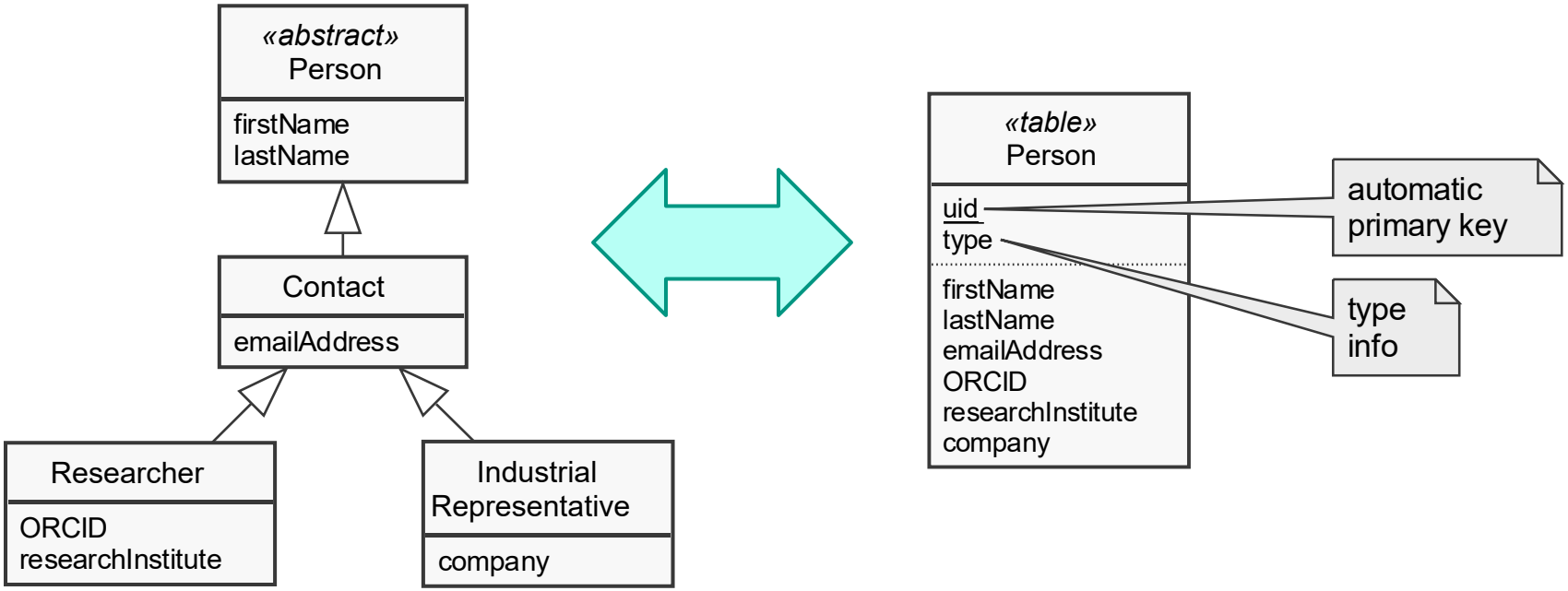
“Represents an inheritance hierarchy of classes as a single table that has columns for all fields of the various classes.”

SINGLE TABLE INHERITANCE

[Fowler]

Single Table Inheritance (pp. 278)

"Represents an inheritance hierarchy of classes as a single table that has columns for all fields of the various classes."



[Fowler]

Discussion

Advantages

- simple database schema: only one table
- no joins required
- refactoring that moves fields up or down do not require database changes

Disadvantages

- direct use of tables confusing because of unused fields
  - depends on actual data characteristics and database abilities to compress empty columns
- tables get very large
  - many indexes & frequent locking
    - ➔ loss of performance
    - solution: introduce additional index tables
- only one namespace for all fields exists
  - for direct accesses tables use naming conventions like "`[Classname]_[Fieldname]`"

[Fowler]

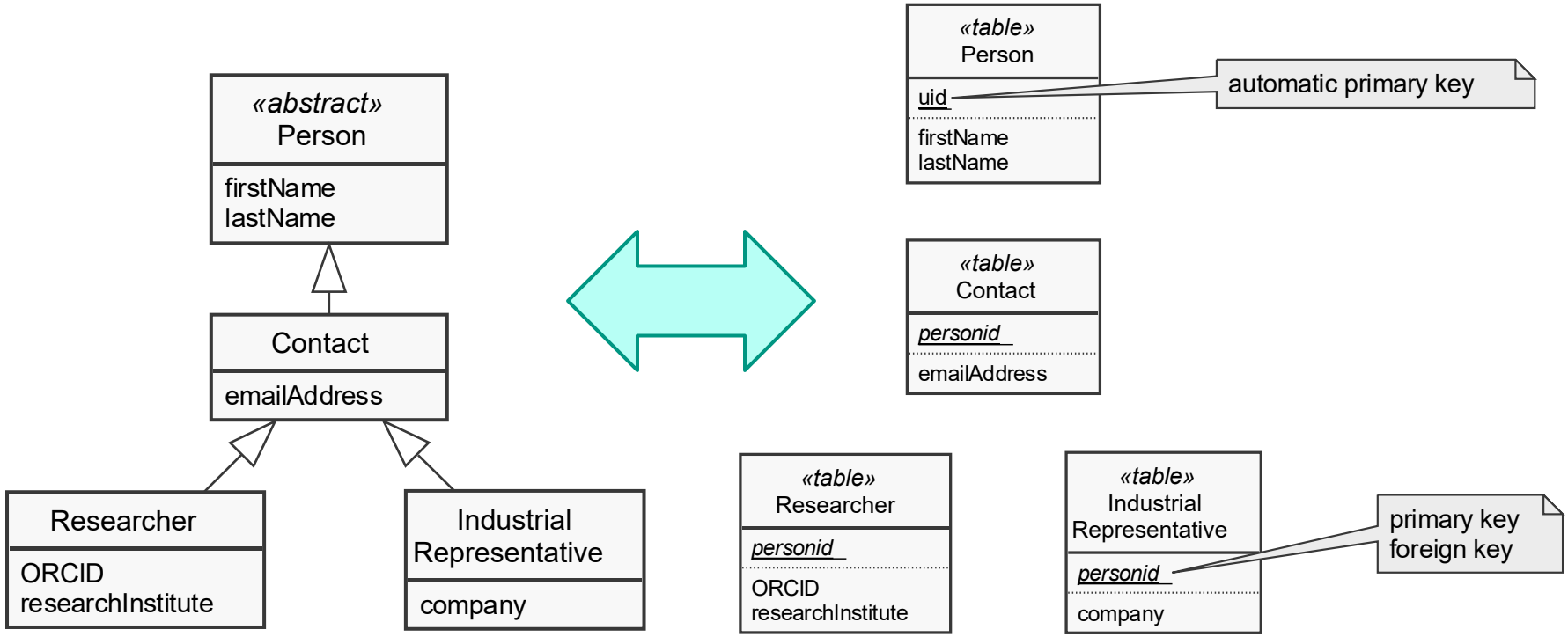
“Represent an inheritance hierarchy of classes with one table for each class”

CLASS TABLE INHERITANCE

[Fowler]

Class Table Inheritance (pp. 285)

"Represent an inheritance hierarchy of classes with one table for each class"



[Fowler]

Discussion

Advantages

- all columns relevant for every row
  - tables are easier to understand
  - tables do not waste space
- easier to map a legacy, pre-existing schema using this strategy
- straightforward relationship between domain model and database schema

Disadvantages

- loading an object in most cases means touching multiple tables
  - Joining multiple tables, multiple queries
    → Performance decreases
- refactoring: Moving fields up and down in the hierarchy implies changing the database schema
- supertypes can become bottlenecks
  - has to be accessed frequently
- high normalisation makes schema harder to understand

[Fowler]

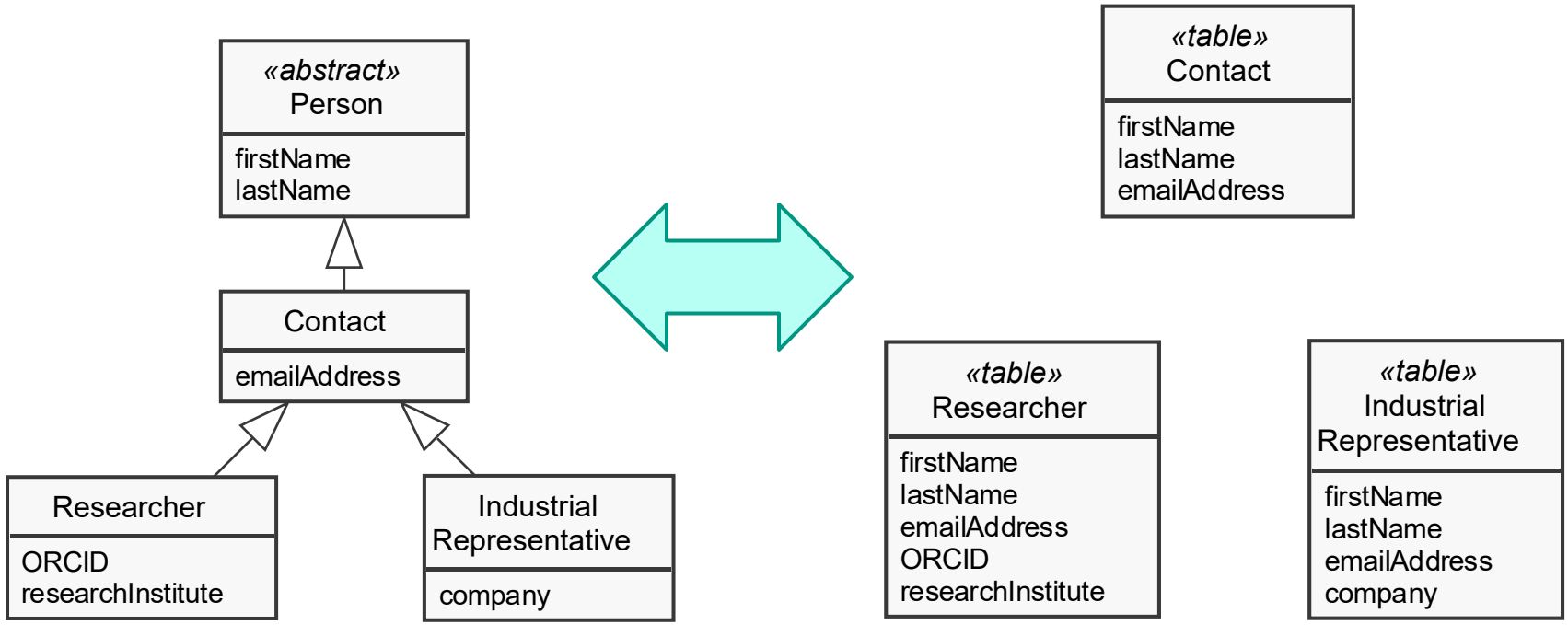
“Represent an inheritance hierarchy of classes with one table per concrete class in the hierarchy”

CONCRETE TABLE INHERITANCE

[Fowler]

Concrete Table Inheritance (pp. 293)

"Represent an inheritance hierarchy of classes with one table per concrete class in the hierarchy"



[Fowler]

Discussion

Advantages

- each table is self-contained; no irrelevant fields
  - useful, if applications directly access the database
- no joins required when reading from concrete mappers
  - concrete subclasses
- only local access to tables can increase performance
  - Each table is usually accessed by one domain object only

Disadvantages

- refactoring: Pushing fields up / down requires database schema changes
- changes to field in supertypes influence all subtype tables
- query on superclass forces check on all subclasses (subtables) or complex join

[Fowler]

When to use what?

[Fowler]

- All patterns intended for a Domain Model with Data Mapper
- Tradeoffs: Duplication of data vs. structure & speed

Single Table Inheritance	Class Table Inheritance	Concrete Table Inheritance
+ Whole hierarchy in one table, no joins required – Wasted space for empty columns, bottleneck for access	+ Clear (normalized) structure – Bad performance, needs joins	+ No joins required – Find requires multiple queries – Changes to superclass affect all tables

➔ Mix patterns in a hierarchy depending on how usual access is
- start with single table inheritance, add others as needed

➔ Consider to use commercial tools
- do not reinvent the wheel, you want to focus on business logic
- still, you need to know the patterns to understand their implications

[Larman]

Inheritance Strategies in Practice

JAVA PERSISTENCE API

Java Persistence API

- POJO persistence API of Java EE
- During the development phase JPA was part of the EJB 3.0 Entity Bean specification: Container Managed Persistence
  - Has much in common with Hibernate
    - Hibernate was *"sponsor"* of several ideas
- Provides full object/relational mapping specification to define mapping between Java objects and a relational database
- Supports use of Java language metadata annotations and/or XML descriptors

[JPAPI]

Entities

O/R Mapping in Hibernate/EJB3 works with relatively simple annotations

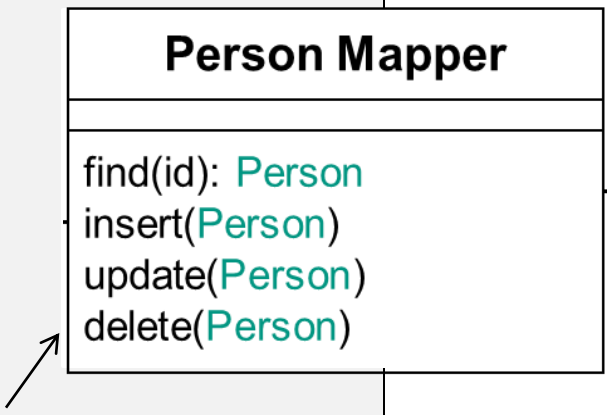
```
@Entity
@Table(name=„Person")  // optional, properties of table
public class Person implements Serializable {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
   // strategy to be used for generating values of
   // the primary key (id herte)
  private Long employeeId;
  private Integer age;
  private String firstname;
  @Column(name="lastname") // optional, to further specify column properties
  private String lastname;

  public Employee() { /*…*/ }

  public Employee(String firstname, String lastname, Date birthdate, String phone) {
      // …
   }

  // Getter and Setter methods
```



EntityManager gets injected here.

Inheritance Strategies

- Defined by annotation
  `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`

```
@Entity
@Table(name=„Person")  // optional
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Person implements Serializable {
 …
}
```

- Supported Strategies:
  - Single Table (Single Table Inheritance)
  - Table per Class (Concrete Table Inheritance)
  - Joined (Class Table Inheritance)
  - *Implicit Polymorphism (in Hibernate)*

e.g. als see http://stackoverflow.com/questions/1373294/inheritance-in-hibernate-annotations

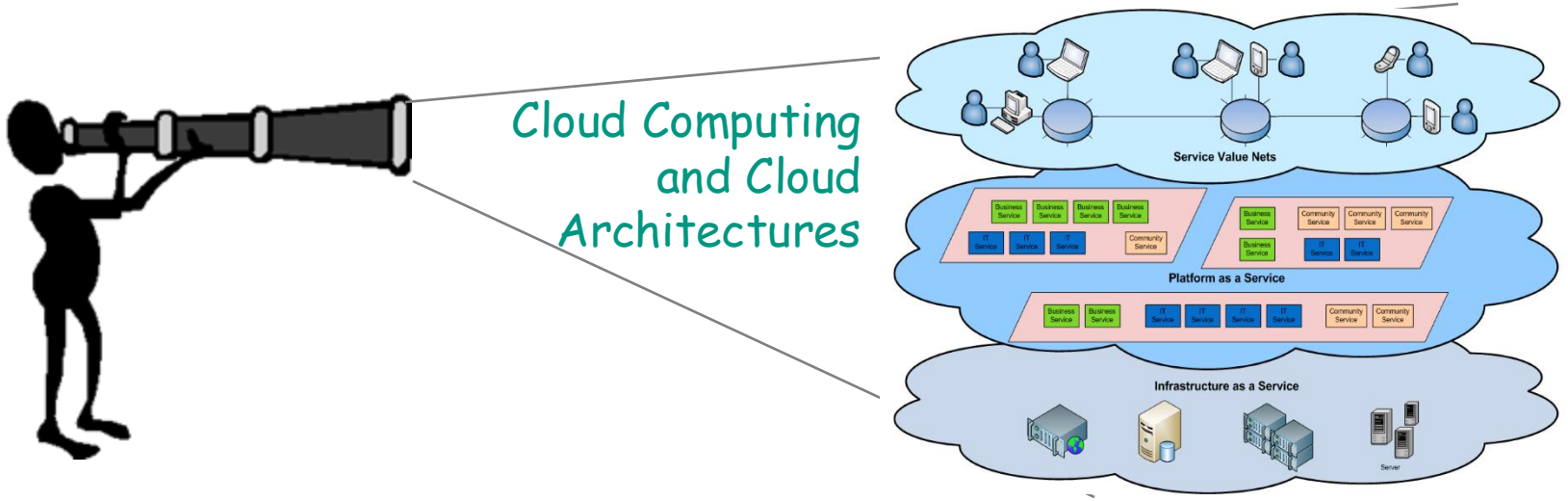
There is More

- [Fowler] lists 52 patterns – this lecture introduced just a few of them
- There are a lot more interesting patterns
  - Query Object, Client / Server Session State, Service Stub, Registry, Lazy Loading, Remote Façade, …
  - ➔ *know where to look up details on them*
- In practice, a lot of patterns can be found in well-known frameworks
  - Eclipse
    - Lazy Loading, Registry, …
  - EJB Session Beans
    - Session State, Registry, Transaction Script, Identity Map, Remote Façade, …

Conclusion

- Broad range of enterprise applications
  - Challenge: manage business logic and data access
- Challenges addressed by different pattern families
  - Domain logic patterns
  - Data source architectural patterns
  - O/R mapping structural patterns

➔ Choose patterns carefully based on system at hand



References (1)

- [Fowler] Martin Fowler, “Patterns of Enterprise Application Architecture”, Addison-Wesley, Boston, 2002.
  - Online resources: http://martinfowler.com/eaaCatalog/
  - Online excerpts from the book on some pattern families
    - Data Source Architecture http://www.informit.com/articles/article.aspx?p=1398618
    - Domain Logic: http://www.informit.com/articles/article.aspx?p=1398617
    - O/R Mapping: http://www.informit.com/articles/article.aspx?p=30661&seqNum=4
- [Alto] Altova, “MapForce Integration with Visual Studio .NET”, last retrieved 2018-12-04 http://www.altova.com/visual_studio_integration.html
- [Brewer 2012] Brewer, Eric. "CAP twelve years later: How the" rules" have changed." *Computer* 45, no. 2 (2012): 23-29.
- [JPAPI] Oracle, “Java Persistence API FAQ”, last retrieved 2018-12-04 http://www.oracle.com/technetwork/java/javaee/persistence-jsp-136066.html
- [MIT CISR 2007] MIT Center for Information Systems Research, Peter Weill, Director, as presented at the Sixth e-Business Conference, Barcelona Spain, March 27, 2007 (https://en.wikipedia.org/wiki/Enterprise_architecture#cite_note-4)
- [Smi01] C. U. Smith and L. G. Williams, “Performance Solutions: Practical Guide to Creating Responsive, Scalable Software”, Addison-Wesley, Boston, 2001

https://martinfowler.com/articles/microservices.html

Further Reading:

http://www.se-radio.net/2015/01/episode-217-james-turnbull-on-docker/

Hasselbring and Steinacker, Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, 2017

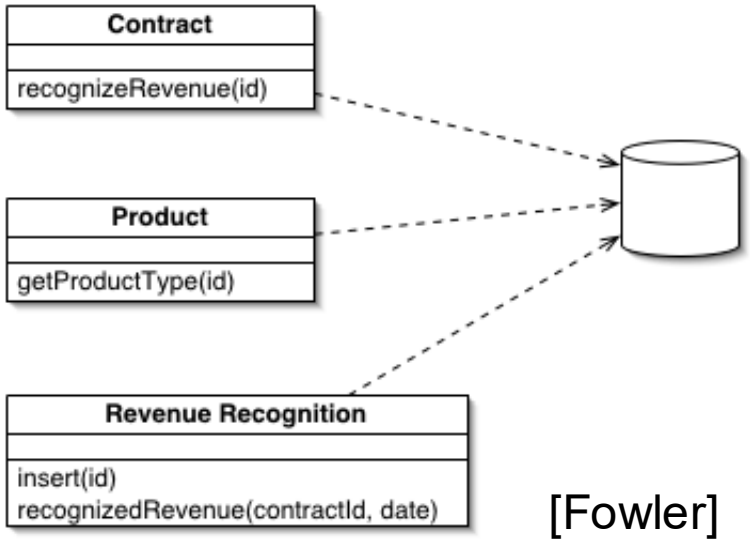
Other Patterns

APPENDIX

Table Module (pp. 125)

"A single instance that handles the business logic for all rows in a database table or view."

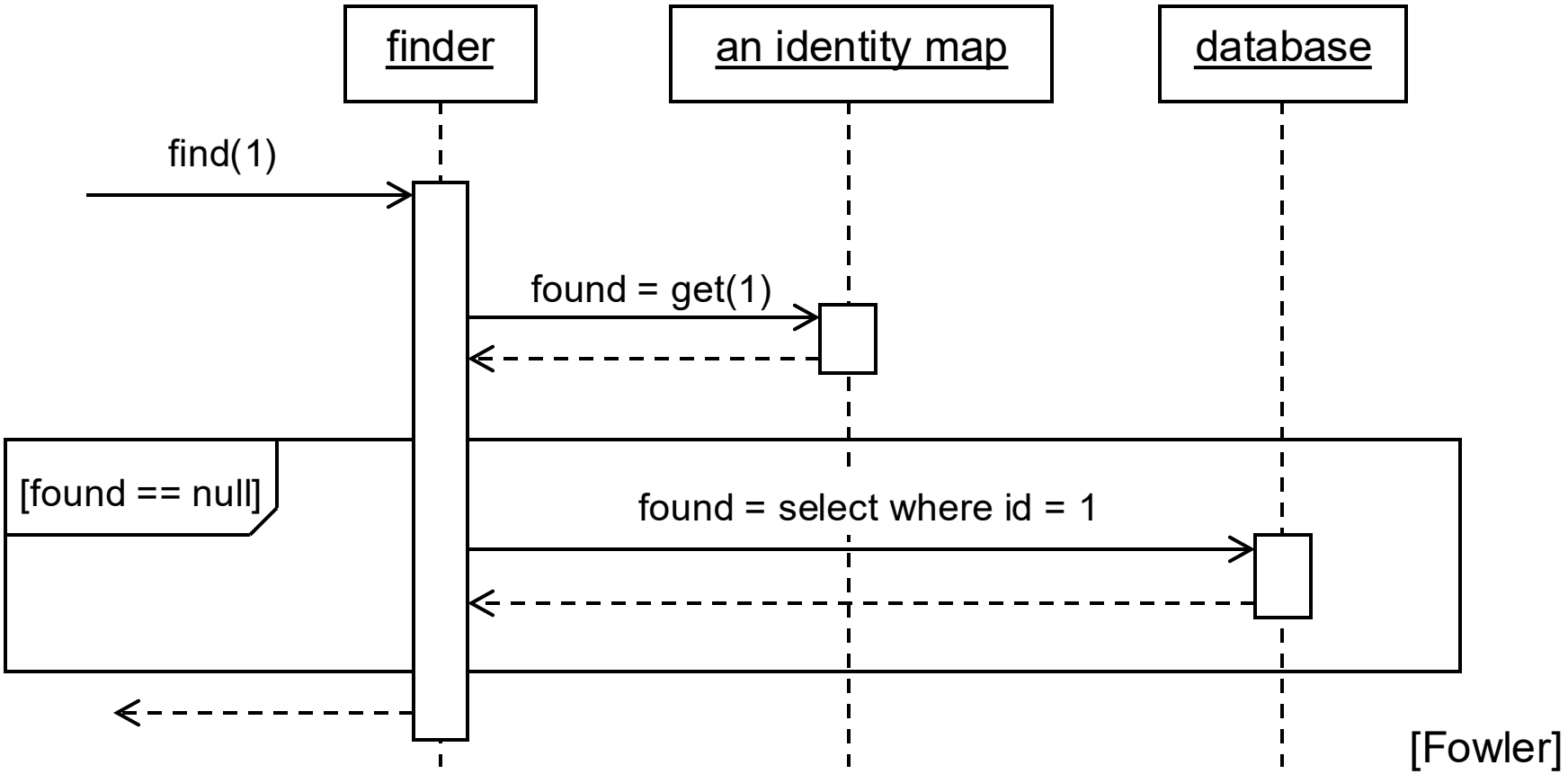
- Table Module organizes domain logic with one class per table
  - a single(ton) instance of the class contains various procedures that will act on data
    - to distinguish on which data set to operate, you need to pass in an identifier
- Primary distinction from Domain Model (pp. 116), consider many orders exist
  - Domain Model will have one order object per order while
  - Table Module will have one order object to handle all orders.



[Fowler]

Identity Map (pp. 195)

"Ensure that each object gets loaded only once by keeping every loaded object in a map. Look up objects using the map when referring to them."



Association Table Mapping (pp. 248)

„Saves an association as a table with foreign keys to the tables that are linked by the association“